

The background is a composite image. It features a world map with a focus on Europe and Africa. Overlaid on the map is a close-up of a hand typing on a vintage, light-colored keyboard. In the upper right, there is a mechanical component, likely from a typewriter, showing a key's internal mechanism. The text is overlaid in a large, white, sans-serif font.

C++ Programming Language for Beginners with Easy tips.

MALINI DEVI J

C++ Programming Language for Beginners with Easy tips.



- MALINI DEVI J

C++ is a middle-level programming language developed by Bjarne Stroustrup starting in 1979 at Bell Labs. C++ runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX.

Audience

This tutorial has been prepared for the beginners to help them understand the basic to advanced concepts related to C++.

Copyright & Disclaimer

Copyright 2014 by MALINI DEVI J.

All the content and graphics published in this e-book are the property of MALINI DEVI J. The user of this e-book is prohibited to reuse, retain, copy, distribute or republish any contents or a part of contents of this e-book in any manner without written consent of the publisher.

1. OVERVIEW 1

Object-Oriented Programming 1

Standard Libraries 1

The ANSI Standard 1

Learning C++ 2

Use of C++ 2

2. ENVIORNMENT SETUP 3

Try it Option Online 3

Local Environment Setup 3

Installing GNU C/C++ Compiler: 4

3. BASIC SYNTAX 6

C++ Program Structure: 6

Compile & Execute C++ Program: 7

Semicolons & Blocks in C++ 7

C++ Identifiers 8

C++ Keywords 8

Trigraphs 9

Whitespace in C++ 10

4. COMMENTS IN C++ 11

5. DATA TYPES 13

Primitive Built-in Types 13

typedef Declarations 15

	Enumerated Types	16
6. VARIABLE TYPES		17
	Variable Definition in C++	17
	Variable Declaration in C++	18
	Lvalues and Rvalues	20
7. VARIABLE SCOPE		21
	Local Variables	21
	Global Variables	22
	Initializing Local and Global Variables	23
8. CONSTANTS/LITERALS		24
	Integer Literals	24
	Floating-point Literals	24
	Boolean Literals	25
	Character Literals	25
	String Literals	26
	Defining Constants	27
9. MODIFIER TYPES		29
	Type Qualifiers in C++	30
10. STORAGE CLASSES		31
	The auto Storage Class	31
	The register Storage Class	31
	The static Storage Class	31
	The extern Storage Class	33
	The mutable Storage Class	34
11. OPERATORS		35

Arithmetic Operators	35
Relational Operators	37
Logical Operators	40
Bitwise Operators	41
Assignment Operators	44
Misc Operators	47
Operators Precedence in C++	48

12. LOOP TYPES 51

While Loop	52
for Loop	54
do...while Loop	56
nested Loops	58
Loop Control Statements	60
Break Statement	61
continue Statement	63
goto Statement	65
The Infinite Loop	67

13. DECISION-MAKING STATEMENTS 69

If Statement	70
if...else Statement	72
if else if else Statement	73
Switch Statement	75
Nested if Statement	78
The ? : Operator	81

14. FUNCTIONS 82

Defining a Function	82
---------------------	----

Function Declarations	83
Calling a Function	84
Function Arguments	85
Call by Value	86
Call by Pointer	87
Call by Reference	89
Default Values for Parameters	90

15. NUMBERS 93

Defining Numbers in C++	93
Math Operations in C++	94
Random Numbers in C++	96

16. ARRAYS 98

Declaring Arrays	98
Initializing Arrays	98
Accessing Array Elements	99
Arrays in C++	100
Pointer to an Array	103
Passing Arrays to Functions	105
Return Array from Functions	107

17. STRINGS 111

The C-Style Character String	111
The String Class in C++	114

18. POINTERS 116

What are Pointers?	116
Using Pointers in C++	117

Null Pointers	119
Pointer Arithmetic	120
Pointers vs Arrays	124
Array of Pointers	126
Pointer to a Pointer	128
Passing Pointers to Functions	130
Return Pointer from Functions	132



C++

C++ is a statically typed, compiled, general-purpose, case-sensitive, free-form programming language that supports procedural, object-oriented, and generic programming..

C++ is regarded as a **middle-level** language, as it comprises a combination of both high-level and low-level language features.

C++ was developed by Bjarne Stroustrup starting in 1979 at Bell Labs in Murray Hill, New Jersey, as an enhancement to the C language and originally named C with Classes but later it was renamed C++ in 1983.

C++ is a superset of C, and that virtually any legal C program is a legal C++ program.

Note: A programming language is said to use static typing when type checking is performed during compile-time as opposed to run-time.

Object-Oriented Programming

C++ fully supports object-oriented programming, including the four pillars of object-oriented development:

Encapsulation

Data hiding

Inheritance

Polymorphism

Standard Libraries

Standard C++ consists of three important parts:

The core language giving all the building blocks including variables, data types and literals, etc.

The C++ Standard Library giving a rich set of functions manipulating files, strings, etc.

The Standard Template Library (STL) giving a rich set of methods manipulating data structures, etc.

The ANSI Standard

The ANSI standard is an attempt to ensure that C++ is portable; that code you write for Microsoft's compiler will compile without errors, using a compiler on a Mac, UNIX, a Windows box, or an Alpha.

C++

The ANSI standard has been stable for a while, and all the major C++ compiler manufacturers support the ANSI standard.

Learning C++

The most important thing while learning C++ is to focus on concepts.

The purpose of learning a programming language is to become a better programmer; that is, to become more effective at designing and implementing new systems and at maintaining old ones.

C++ supports a variety of programming styles. You can write in the style of Fortran, C, Smalltalk, etc., in any language. Each style can achieve its aims effectively while maintaining runtime and space efficiency.

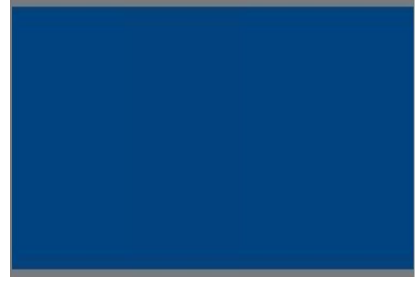
Use of C++

C++ is used by hundreds of thousands of programmers in essentially every application domain.

C++ is being highly used to write device drivers and other software that rely on direct manipulation of hardware under real-time constraints.

C++ is widely used for teaching and research because it is clean enough for successful teaching of basic concepts.

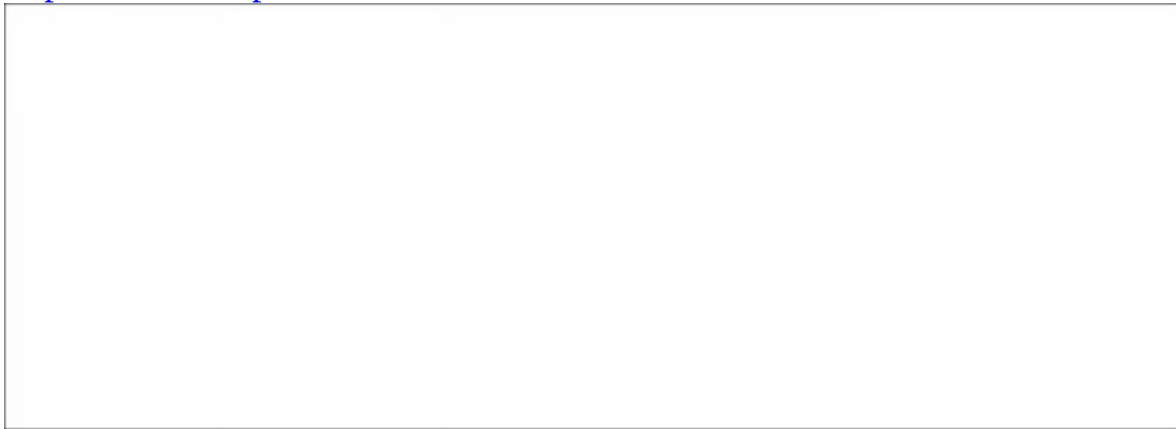
Anyone who has used either an Apple Macintosh or a PC running Windows has indirectly used C++ because the primary user interfaces of these systems are written in C++.



Try it Option Online

You really do not need to set up your own environment to start learning C++ programming language. Reason is very simple, we have already set up C++ Programming environment online, so that you can compile and execute all the available examples online at the same time when you are doing your theory work. This gives you confidence in what you are reading and to check the result with different options. Feel free to modify any example and execute it online.

Try the following example using our online compiler option available at <http://www.compileonline.com/>



```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello World";

    return 0;
}
```

For most of the examples given in this tutorial, you will find **Try it** option in our website code sections at the top right corner that will take you to the online compiler. So just make use of it and enjoy your learning.

Local Environment Setup

If you are still willing to set up your environment for C++, you need to have the following two softwares on your computer.

Text Editor:

This will be used to type your program. Examples of few editors include Windows Notepad, OS Edit command, Brief, Epsilon, EMACS, and vim or vi.

Name and version of text editor can vary on different operating systems. For example, Notepad will be used on Windows and vim or vi can be used on windows as well as Linux, or UNIX.

The files you create with your editor are called source files and for C++ they typically are named with the extension .cpp, .cp, or .c.

A text editor should be in place to start your C++ programming.

C++ Compiler:

This is an actual C++ compiler, which will be used to compile your source code into final executable program.

Most C++ compilers don't care what extension you give to your source code, but if you don't specify otherwise, many will use .cpp by default.

Most frequently used and free available compiler is GNU C/C++ compiler, otherwise you can have compilers either from HP or Solaris if you have the respective Operating Systems.

Installing GNU C/C++ Compiler:

UNIX/Linux Installation:

If you are using **Linux or UNIX** then check whether GCC is installed on your system by entering the following command from the command line:

```
$ g++ -v
```

If you have installed GCC, then it should print a message such as the following:

```
Using built-in specs.
```

```
Target: i386-redhat-linux
```

```
Configured with: ../configure --prefix=/usr .....
```

```
Thread model: posix
```

```
gcc version 4.1.2 20080704 (Red Hat 4.1.2-46)
```

If GCC is not installed, then you will have to install it yourself using the detailed instructions available at <http://gcc.gnu.org/install/> .

Mac OS X Installation:

If you use Mac OS X, the easiest way to obtain GCC is to download the Xcode development environment from Apple's website and follow the simple installation instructions.

Xcode is currently available at developer.apple.com/technologies/tools/.

Windows Installation:

To install GCC at Windows you need to install MinGW. To install MinGW, go to the MinGW homepage, www.mingw.org, and follow the link to the MinGW download page. Download the latest version of the MinGW installation program which should be named MinGW-<version>.exe.

While installing MinGW, at a minimum, you must install gcc-core, gcc-g++, binutils, and the MinGW runtime, but you may wish to install more.

C++

Add the bin subdirectory of your MinGW installation to your **PATH** environment variable so that you can specify these tools on the command line by their simple names.

When the installation is complete, you will be able to run gcc, g++, ar, ranlib, dlltool, and several other GNU tools from the Windows command line.



C++

When we consider a C++ program, it can be defined as a collection of objects that communicate via invoking each other's methods. Let us now briefly look into what a class, object, methods, and instant variables mean.

Object - Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors - wagging, barking, and eating. An object is an instance of a class.

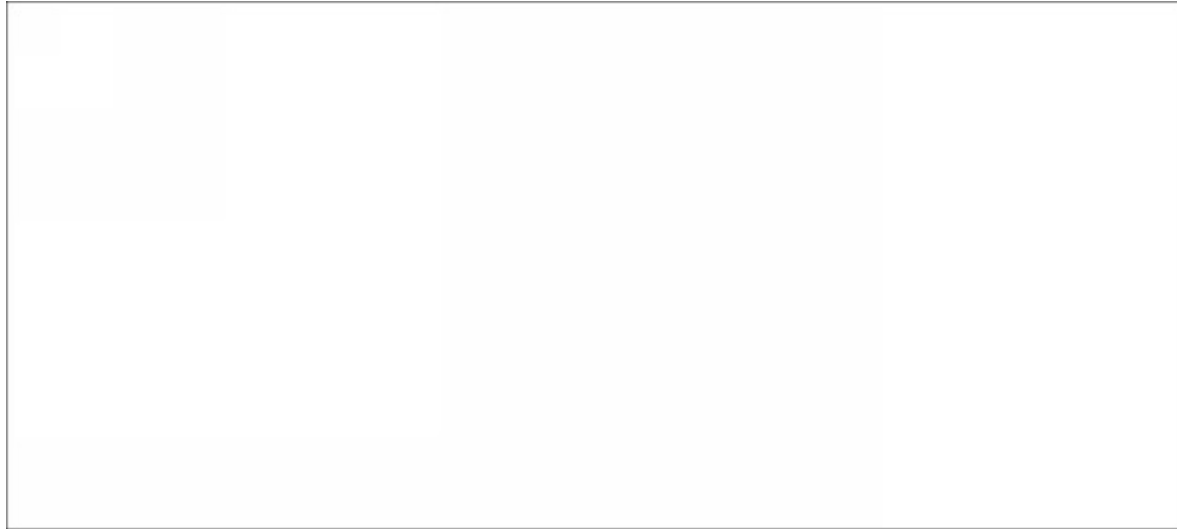
Class - A class can be defined as a template/blueprint that describes the behaviors/states that object of its type support.

Methods - A method is basically a behavior. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed.

Instant Variables - Each object has its unique set of instant variables. An object's state is created by the values assigned to these instant variables.

C++ Program Structure:

Let us look at a simple code that would print the words *Hello World*.



```
#include <iostream>

using namespace std;

// main() is where program execution begins.

int main()
{
    cout << "Hello World"; // prints Hello World return 0;
}
```

Let us look at the various parts of the above program:

The C++ language defines several headers, which contain information that is either necessary or useful to your program. For this program, the header **<iostream>** is needed.

The line **using namespace std;** tells the compiler to use the std namespace. Namespaces are a relatively recent addition to C++.

C++

The next line '**// main() is where program execution begins.**' is a single-line comment available in C++. Single-line comments begin with **//** and stop at the end of the line.

The line **int main()** is the main function where program execution begins.

The next line **cout << "This is my first C++ program.";** causes the message "This is my first C++ program" to be displayed on the screen.

The next line **return 0;** terminates **main()** function and causes it to return the value 0 to the calling process.

Compile & Execute C++ Program:

Let's look at how to save the file, compile and run the program. Please follow the steps given below:

Open a text editor and add the code as above.

Save the file as: **hello.cpp**

Open a command prompt and go to the directory where you saved the file.

Type '**g++ hello.cpp**' and press enter to compile your code. If there are no errors in your code the command prompt will take you to the next line and would generate **a.out** executable file.

Now, type '**a.out**' to run your program.

You will be able to see ' Hello World ' printed

on the window.

```
$ g++ hello.cpp
```

```
$ ./a.out
```

```
Hello World
```

Make sure that g++ is in your path and that you are running it in the directory containing file hello.cpp.

You can compile C/C++ programs using makefile. For more details, you can check our ‘Makefile Tutorial’.

Semicolons & Blocks in C++

In C++, the semicolon is a statement terminator. That is, each individual statement must be ended with a semicolon. It indicates the end of one logical entity.

For example, following are three different statements:

```
x = y;
```

```
y = y+1;
```

C++

```
add(x, y);
```

A block is a set of logically connected statements that are surrounded by opening and closing braces. For example:

```
{  
  
    cout << "Hello World"; // prints Hello World  
    return 0;  
  
}
```

C++ does not recognize the end of the line as a terminator. For this reason, it does not matter where you put a statement in a line. For example:

```
x = y;  
y = y+1;  
add(x, y);
```

is the same as

```
x = y; y = y+1; add(x, y);
```

C++ Identifiers

A C++ identifier is a name used to identify a variable, function, class, module, or any other user-defined item. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores, and digits (0 to 9).

C++ does not allow punctuation characters such as @, \$, and % within

identifiers. C++ is a case-sensitive programming language.

Thus, **Manpower** and **manpower** are two different identifiers in C++.

Here are some examples of acceptable identifiers:

mohd

zara

abc

move_name a_123

myname50

_temp

j

a23b9

retVal

C++ Keywords

The following list shows the reserved words in C++. These reserved words may not be used as constant or variable or any other identifier names.

asm	else	new	this
auto	enum	operator	throw

C++

bool	explicit	private	true
break	export	protected	try
case	extern	public	typedef
catch	false	register	typeid
char	float	reinterpret_cast	typename
class	for	return	union
const	friend	short	unsigned
const_cast	goto	signed	using
continue	if	sizeof	virtual
default	inline	static	void
delete	int	static_cast	volatile
do	long	struct	wchar_t
double	mutable	switch	while
dynamic_cast	namespace	template	

Trigraphs			
-----------	--	--	--

A few characters have an alternative representation, called a trigraph sequence. A trigraph is a three-character sequence that represents a single character and the sequence always starts with two question marks.

Trigraphs are expanded anywhere they appear, including within string literals and character literals, in comments, and in preprocessor directives.

Following are most frequently used trigraph sequences:

--	--

Trigraph

Replacement

C++

??=	#
??/	\
??'	^
??([
??)]
??!	
??<	{
??>	}
??-	~

All the compilers do not support trigraphs and they are not advised to be used because of their confusing nature.

Whitespace in C++

A line containing only whitespace, possibly with a comment, is known as a blank line, and C++ compiler totally ignores it.

Whitespace is the term used in C++ to describe blanks, tabs, newline characters and comments. Whitespace separates one part of a statement from another and enables the compiler to identify where one element in a

statement, such as int, ends and the next element begins. Statement 1:

```
int age;
```

In the above statement there must be at least one whitespace character (usually a space) between int and age for the compiler to be able to distinguish them. Statement 2:

```
fruit = apples + oranges;    // Get the total fruit
```

In the above statement 2, no whitespace characters are necessary between fruit and =, or between = and apples, although you are free to include some if you wish for readability purpose.

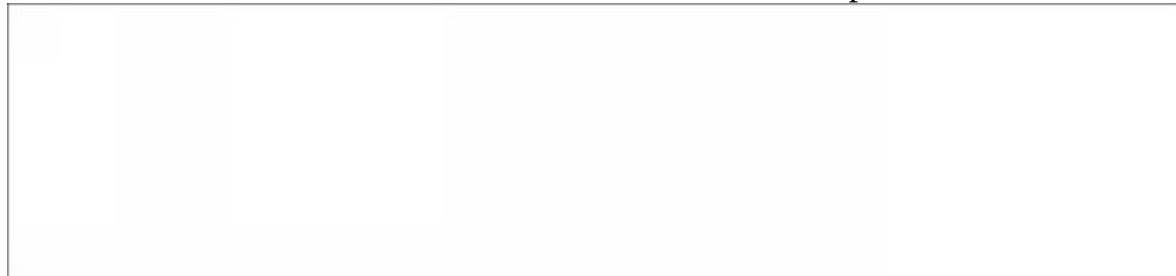


C++

Program comments are explanatory statements that you can include in the C++ code. These comments help anyone reading the source code. All programming languages allow for some form of comments.

C++ supports single-line and multi-line comments. All characters available inside any comment are ignored by C++ compiler.

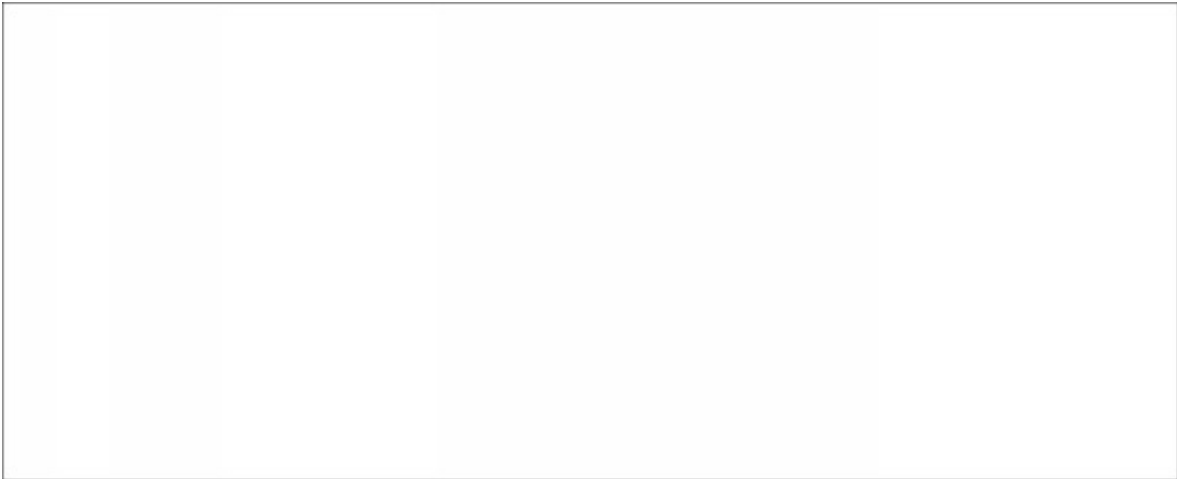
C++ comments start with `/*` and end with `*/`. For example:



```
/* This is a comment */
```

```
/* C++ comments can also  
    span multiple lines  
    */
```

A comment can also start with `//`, extending to the end of the line. For example:



```
#include <iostream>

using namespace std;

main()
{
    cout << "Hello World"; // prints Hello World

    return 0;
}
```

When the above code is compiled, it will ignore // **prints Hello World** and final executable will produce the following result:



Hello World

Within a /* and */ comment, // characters have no special meaning. Within a // comment, /* and */ have no special meaning. Thus, you can "nest" one kind of comment within the other kind. For example:



```
/* Comment out printing of Hello World:
```


C++

```
cout << "Hello World"; // prints Hello World
```

```
*/
```




C++

While writing program in any language, you need to use various variables to store various information. Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

You may like to store information of various data types like character, wide character, integer, floating point, double floating point, boolean etc. Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory.

Primitive Built-in Types

C++ offers the programmer a rich assortment of built-in as well as user defined data types. Following table lists down seven basic C++ data types:

Type	Keyword
Boolean	bool
Character	char
Integer	int

Floating point	float
Double floating point	double
Valueless	void
Wide character	wchar_t

Several of the basic types can be modified using one or more of these type modifiers:

signed

unsigned

short

long

C++

The following table shows the variable type, how much memory it takes to store the value in memory, and what is maximum and minimum value which can be stored in such type of variables.

Type	Typical Bit Width	Typical Range

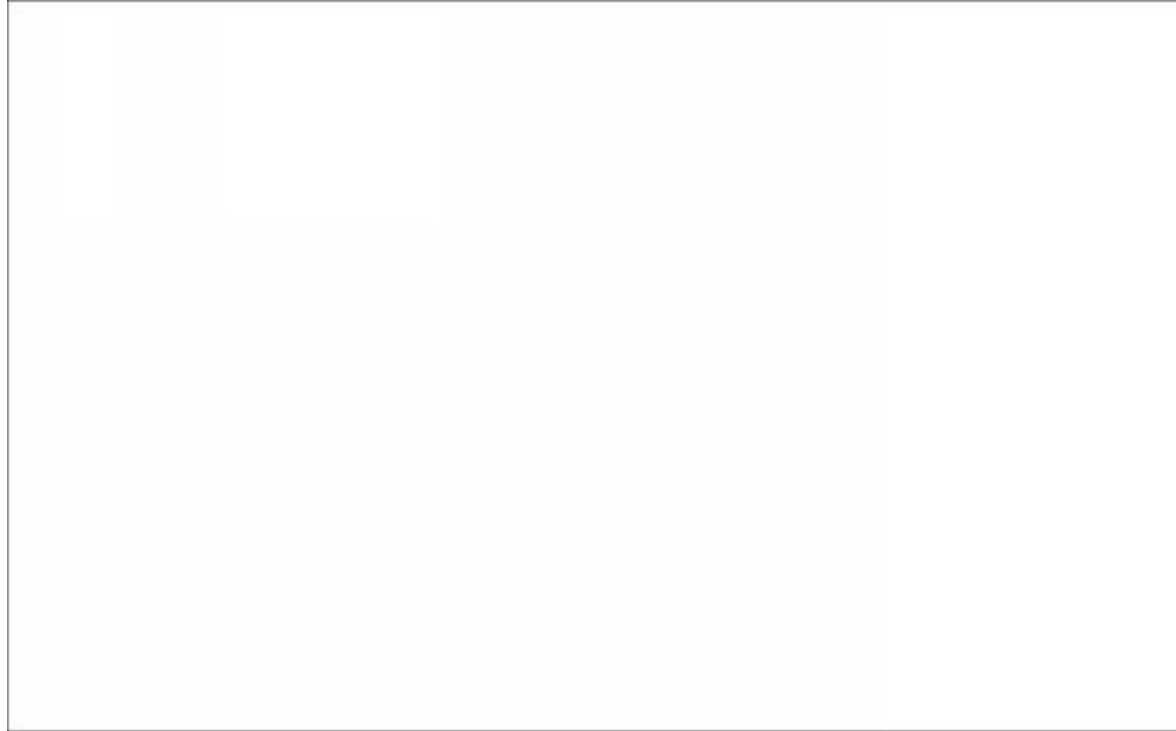
char	1byte	-127 to 127 or 0 to 255
unsigned char	1byte	0 to 255
signed char	1byte	-127 to 127
int	4bytes	-2147483648 to 2147483647
unsigned int	4bytes	0 to 4294967295
signed int	4bytes	-2147483648 to 2147483647
short int	2bytes	-32768 to 32767
unsigned short int	Range	0 to 65,535
signed short int	Range	-32768 to 32767
long int	4bytes	-2,147,483,647 to 2,147,483,647
signed long int	4bytes	same as long int
unsigned long int	4bytes	0 to 4,294,967,295

float	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	8bytes	+/- 1.7e +/- 308 (~15 digits)
wchar_t	2 or 4 bytes	1 wide character

The size of variables might be different from those shown in the above table, depending on the compiler and the computer you are using.

C++

Following is the example, which will produce correct size of various data types on your computer.



```
#include <iostream>

using namespace std;

int main()
{
    cout << "Size of char : " << sizeof(char) << endl;
    cout << "Size of int : " << sizeof(int) << endl;
    cout << "Size of short int : " << sizeof(short int) << endl;
    cout << "Size of long int : " << sizeof(long int) << endl;
```

```
        cout << "Size of float : " << sizeof(float) << endl;

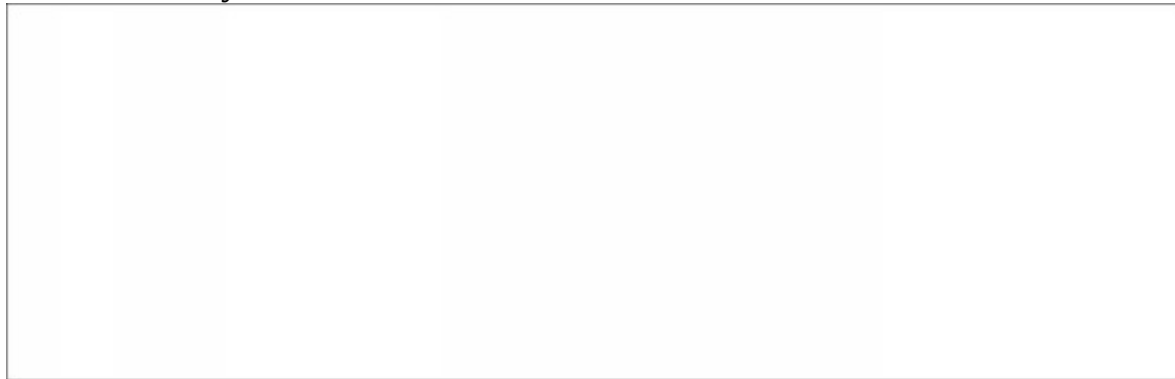
        cout << "Size of double : " << sizeof(double) << endl;
        cout << "Size of wchar_t : " << sizeof(wchar_t) <<
        endl; return 0;

    }
```

This example uses **endl**, which inserts a new-line character after every line and

operator is being used to pass multiple values out to the screen. We are also using **sizeof()** function to get size of various data types.

When the above code is compiled and executed, it produces the following result which can vary from machine to machine:



Size of char : 1

Size of int : 4

Size of short int : 2

Size of long int : 4

Size of float : 4

Size of double : 8

Size of wchar_t : 4

typedef Declarations

You can create a new name for an existing type using **typedef**. Following is the simple syntax to define a new type using typedef:

`typedef type newname;`

For example, the following tells the compiler that `feet` is another name for `int`:

C++

```
typedef int feet;
```

Now, the following declaration is perfectly legal and creates an integer variable called distance:

```
feet distance;
```

Enumerated Types

An enumerated type declares an optional type name and a set of zero or more identifiers that can be used as values of the type. Each enumerator is a constant whose type is the enumeration.

Creating an enumeration requires the use of the keyword **enum**. The general form of an enumeration type is:

```
enum enum-name { list of names } var-list;
```

Here, the enum-name is the enumeration's type name. The list of names is comma separated.

For example, the following code defines an enumeration of colors called colors and the variable c of type color. Finally, c is assigned the value "blue".

```
enum color { red, green, blue } c;
```

```
c = blue;
```

By default, the value of the first name is 0, the second name has the value 1, and the third has the value 2, and so on. But you can give a name, a specific value by adding an initializer. For example, in the following enumeration, **green** will have

the value 5.

--

```
enum color { red, green=5, blue };
```

Here, **blue** will have a value of 6 because each name will be one greater than the one that precedes it.



C++

A variable provides us with named storage that our programs can manipulate. Each variable in C++ has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C++ is case-sensitive:

There are following basic types of variable in C++ as explained in last chapter:

Type	Description
bool	Stores either value true or false.
char	Typically a single octet (one byte). This is an integer type.
int	The most natural size of integer for the machine.

float	A single-precision floating point value.
double	A double-precision floating point value.
void	Represents the absence of type.
wchar_t	A wide character type.

C++ also allows to define various other types of variables, which we will cover in subsequent chapters like **Enumeration**, **Pointer**, **Array**, **Reference**, **Data structures**, and **Classes**.

Following section will cover how to define, declare and use various types of variables.

Variable Definition in C++

A variable definition tells the compiler where and how much storage to create for the variable. A variable definition specifies a data type, and contains a list of one or more variables of that type as follows:

C++

```
type variable_list;
```

Here, **type** must be a valid C++ data type including char, w_char, int, float, double, bool or any user-defined object, etc., and **variable_list** may consist of one or more identifier names separated by commas. Some valid declarations are shown here:

```
int    i, j, k;
```

```
char    c, ch;
```

```
float    f, salary;
```

```
double d;
```

The line **int i, j, k;** both declares and defines the variables i, j and k; which instructs the compiler to create variables named i, j and k of type int.

Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as follows:

```
type variable_name = value;
```

Some examples are:



extern int	d = 3, f = 5;	// declaration of d and f.
int d = 3,	f = 5;	// definition and initializing d and f.
byte z	= 22;	// definition and initializes z.
char x	= 'x';	// the variable x has the value 'x'.

For definition without an initializer: variables with static storage duration are implicitly initialized with NULL (all bytes have the value 0); the initial value of all other variables is undefined.

Variable Declaration in C++

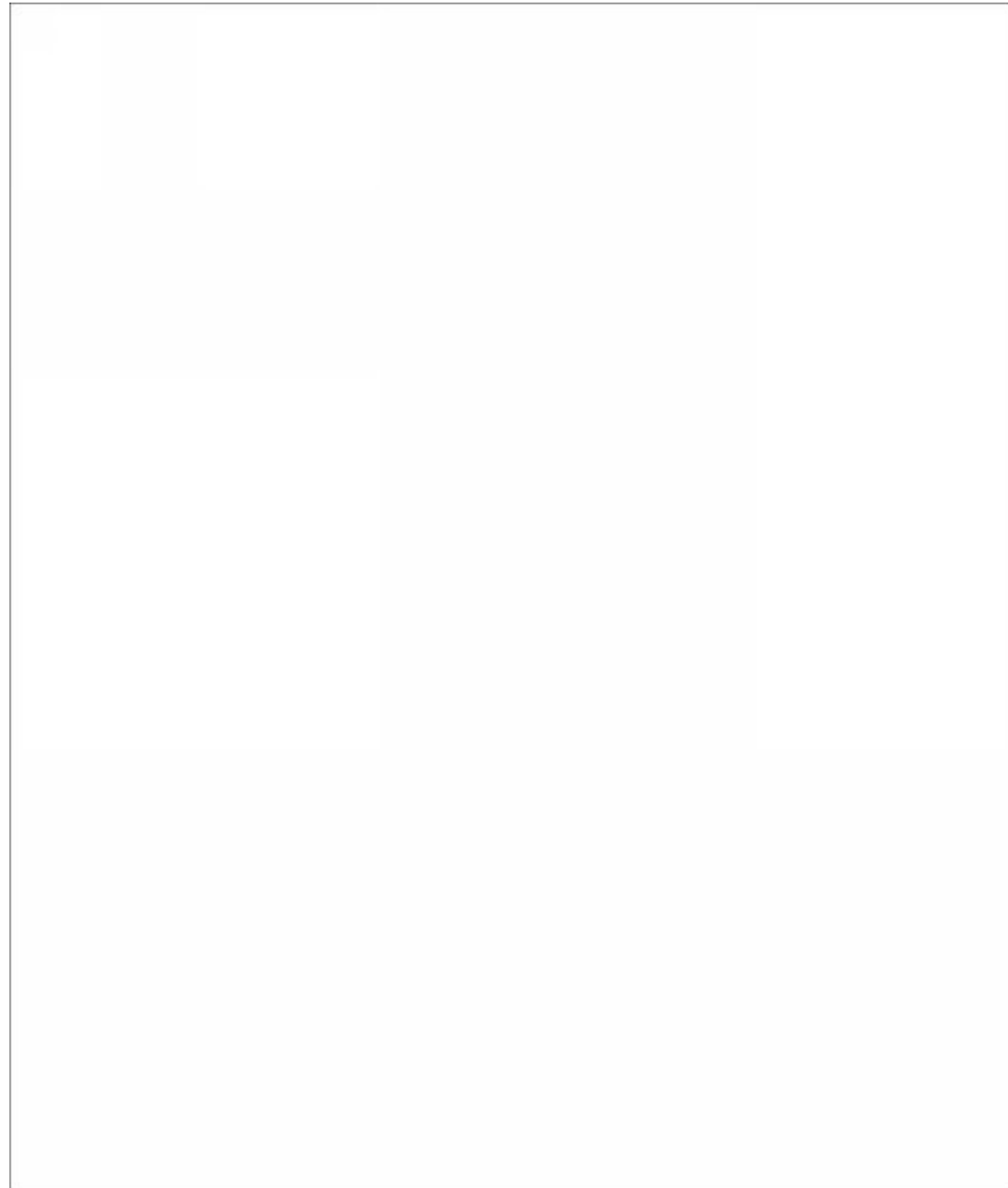
A variable declaration provides assurance to the compiler that there is one variable existing with the given type and name so that compiler proceed for further compilation without needing complete detail about the variable. A variable declaration has its meaning at the time of compilation only, compiler needs actual variable declaration at the time of linking of the program.

A variable declaration is useful when you are using multiple files and you define your variable in one of the files which will be available at the time of linking of the program. You will use **extern** keyword to declare a variable at any place. Though you can declare a variable multiple times in your C++ program, but it can be defined only once in a file, a function or a block of code.

Example:

C++

Try the following example where a variable has been declared at the top, but it has been defined inside the main function:



```
#include <iostream>
```

```
using namespace std;
```

```
    Variable declaration: extern int a, b; extern int c;
```

```
extern float f;
```

```
int main ()
```

```
{
```

```
    Variable definition: int a, b;
```

```
    int c; float f;
```

```
        actual initialization a = 10;
```

```
        b = 20;
```

```
        c = a + b;
```

```
        cout << c << endl ;
```

```
        f = 70.0/3.0;
```

```
        cout << f << endl ;
```

```
        return 0;
```

```
}
```

When the above code is compiled and executed, it produces the following result:

--

30

23.3333

19

C++

Same concept applies on function declaration where you provide a function name at the time of its declaration and its actual definition can be given anywhere else. For example:



function declaration `int func();`

```
int main()
```

```
{
```

```
    function call int i = func();
```

```
}
```

function definition `int func()`

```
{
```

```
    return 0;
```

}

Lvalues and Rvalues

There are two kinds of expressions in C++:

lvalue : Expressions that refer to a memory location is called "lvalue" expression. An lvalue may appear as either the left-hand or right-hand side of an assignment.

rvalue : The term rvalue refers to a data value that is stored at some address in memory. An rvalue is an expression that cannot have a value assigned to it which means an rvalue may appear on the right- but not left-hand side of an assignment.

Variables are lvalues and so may appear on the left-hand side of an assignment. Numeric literals are rvalues and so may not be assigned and cannot appear on the left-hand side. Following is a valid statement:

```
int g = 20;
```

But the following is not a valid statement and would generate compile-time error:

```
10 = 20;
```



C++

A scope is a region of the program and broadly speaking there are three places, where variables can be declared:

Inside a function or a block which is called local variables,

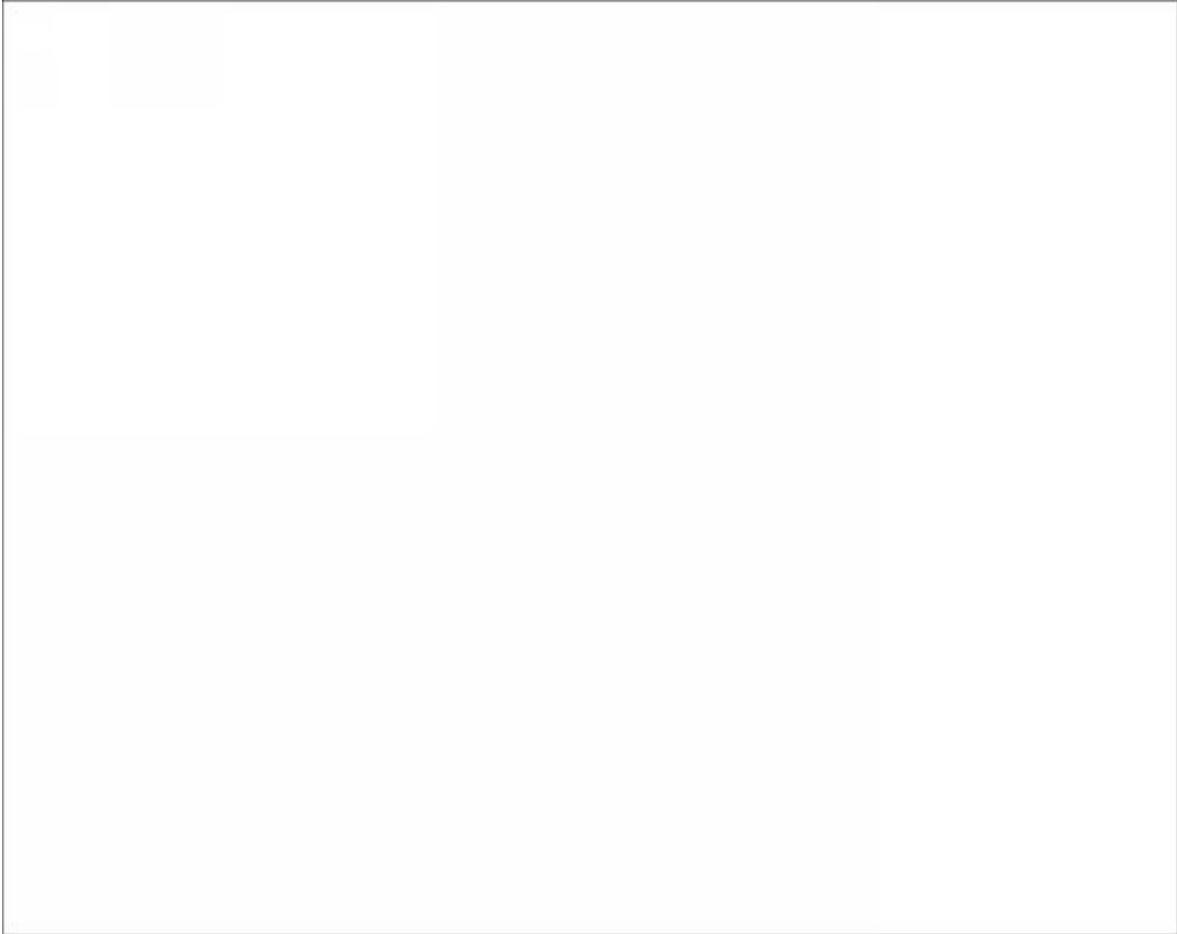
In the definition of function parameters which is called formal parameters.

Outside of all functions which is called global variables.

We will learn what a function is, and its parameter in subsequent chapters. Here let us explain what local and global variables are.

Local Variables

Variables that are declared inside a function or block are local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. Following is the example using local variables:



```
#include <iostream>
```

```
using namespace std;
```

```
int main ()
```

```
{
```

```
    Local variable declaration: int a, b;
```

```
    int c;
```

```
        actual initialization
```

```
        a = 10;
```

```
        b = 20;
```

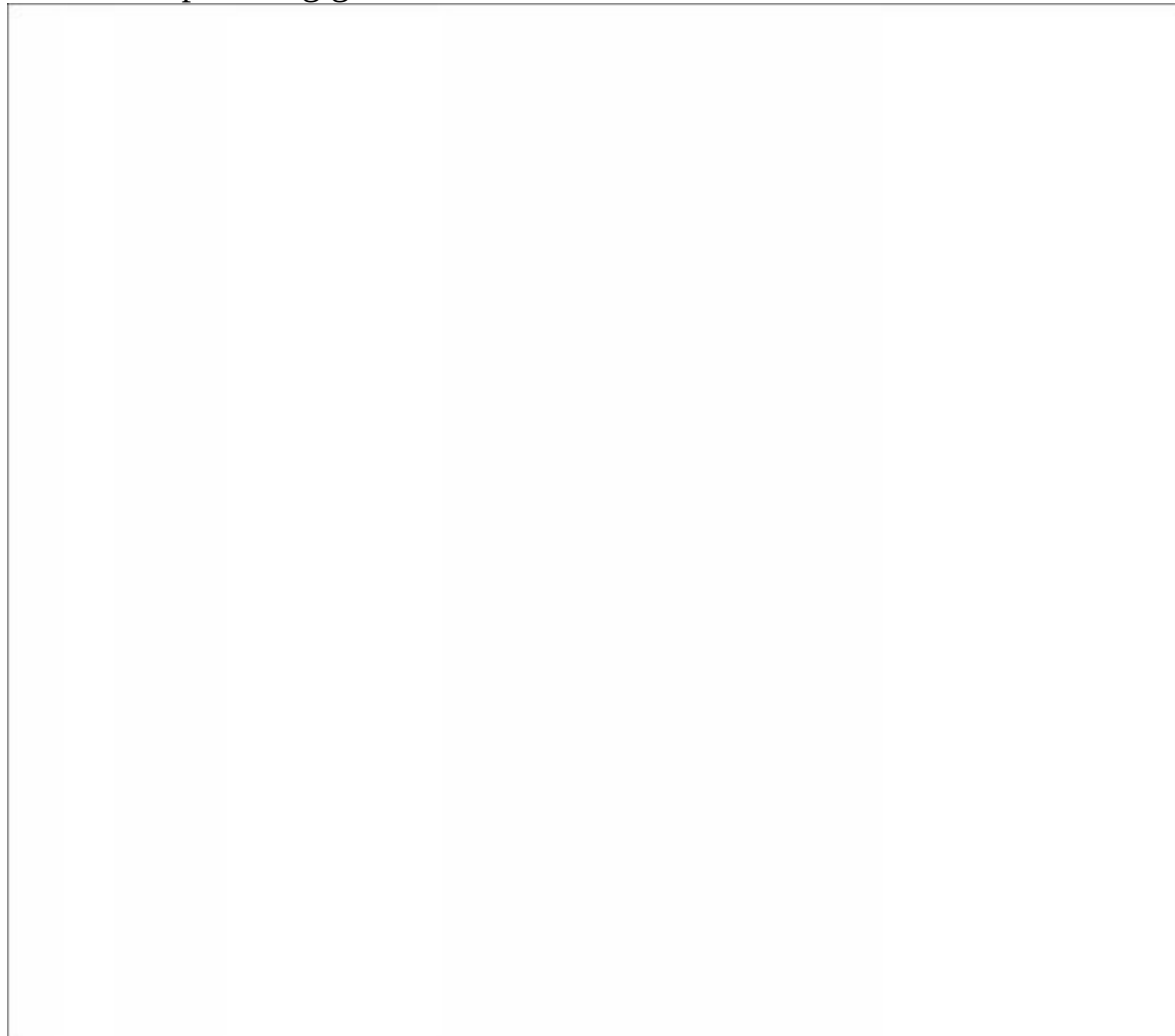
```
    c = a + b;  
  
    cout << c;  
  
    return 0;  
}
```

C++

Global Variables

Global variables are defined outside of all the functions, usually on top of the program. The global variables will hold their value throughout the life-time of your program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. Following is the example using global and local variables:



```
#include <iostream>
```

```
using namespace std;
```

Global variable declaration: int g;

```
int main ()
```

```
{
```

Local variable declaration: int a, b;

actual initialization

```
a = 10;
```

```
b = 20;
```

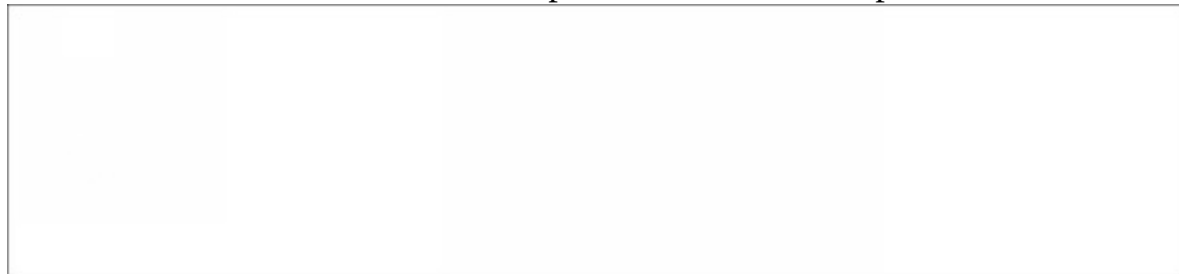
```
g = a + b;
```

```
cout << g;
```

```
return 0;
```

```
}
```

A program can have same name for local and global variables but value of local variable inside a function will take preference. For example:



```
#include <iostream>
```

```
using namespace std;
```

Global variable declaration: int g = 20;

C++

```
int main ()  
{  
    Local variable declaration: int g = 10;  
  
    cout << g;  
  
    return 0;  
}
```

When the above code is compiled and executed, it produces the following result:

```
10
```

Initializing Local and Global Variables

When a local variable is defined, it is not initialized by the system, you must initialize it yourself. Global variables are initialized automatically by

Data Type	Initializer

int	0
char	'\0'
float	0
double	0
pointer	NULL

It is a good programming practice to initialize variables properly, otherwise sometimes program would produce unexpected result.



C++

Constants refer to fixed values that the program may not alter and they are called **literals**.

Constants can be of any of the basic data types and can be divided into Integer Numerals, Floating-Point Numerals, Characters, Strings and Boolean Values.

Again, constants are treated just like regular variables except that their values cannot be modified after their definition.

Integer Literals

An integer literal can be a decimal, octal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and nothing for decimal.

An integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order.

Here are some examples of integer literals:

--

212	// Legal
215u	// Legal
0xFeeL	// Legal
078	// Illegal: 8 is not an octal digit
032UU	// Illegal: cannot repeat a suffix

Following are other examples of various types of Integer literals:

--

85	// decimal
0213	// octal
0x4b	// hexadecimal
30	// int
30u	// unsigned int
30l	// long
30ul	// unsigned long

Floating-point Literals

A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part. You can represent floating point literals either in decimal form or exponential form.

C++

While representing using decimal form, you must include the decimal point, the exponent, or both and while representing using exponential form, you must include the integer part, the fractional part, or both. The signed exponent is introduced by e or E.

Here are some examples of floating-point literals:



3.14159	// Legal
314159E-5L	// Legal
510E	// Illegal: incomplete exponent
210f	// Illegal: no decimal or exponent
.e55	// Illegal: missing integer or fraction

Boolean Literals

There are two Boolean literals and they are part of standard C++ keywords:

A value of **true** representing true.

A value of **false** representing false.

You should not consider the value of true equal to 1 and value of false equal to 0.

Character Literals

Character literals are enclosed in single quotes. If the literal begins with L (uppercase only), it is a wide character literal (e.g., L'x') and should be stored in **wchar_t** type of variable. Otherwise, it is a narrow character literal (e.g., 'x') and can be stored in a simple variable of **char** type.

A character literal can be a plain character (e.g., 'x'), an escape sequence (e.g., '\t'), or a universal character (e.g., '\u02C0').

There are certain characters in C++ when they are preceded by a backslash they will have special meaning and they are used to represent like newline (\n) or tab (\t). Here, you have a list of some of such escape sequence codes:

Escape sequence	Meaning
\\	\ character
\'	' character
\"	" character
\?	? character

C++

\a	Alert or bell
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\ooo	Octal number of one to three digits
\xhh . . .	Hexadecimal number of one or more digits

Following is the example to show a few escape sequence characters:

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello\tWorld\n\n";
    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

Hello World

String Literals

String literals are enclosed in double quotes. A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters.

C++

You can break a long line into multiple lines using string literals and separate them using whitespaces.

Here are some examples of string literals. All the three forms are identical strings.

"hello, dear"

"hello, \

dear"

"hello, " "d" "ear"

Defining Constants

There are two simple ways in C++ to define constants:

Using **#define** preprocessor.

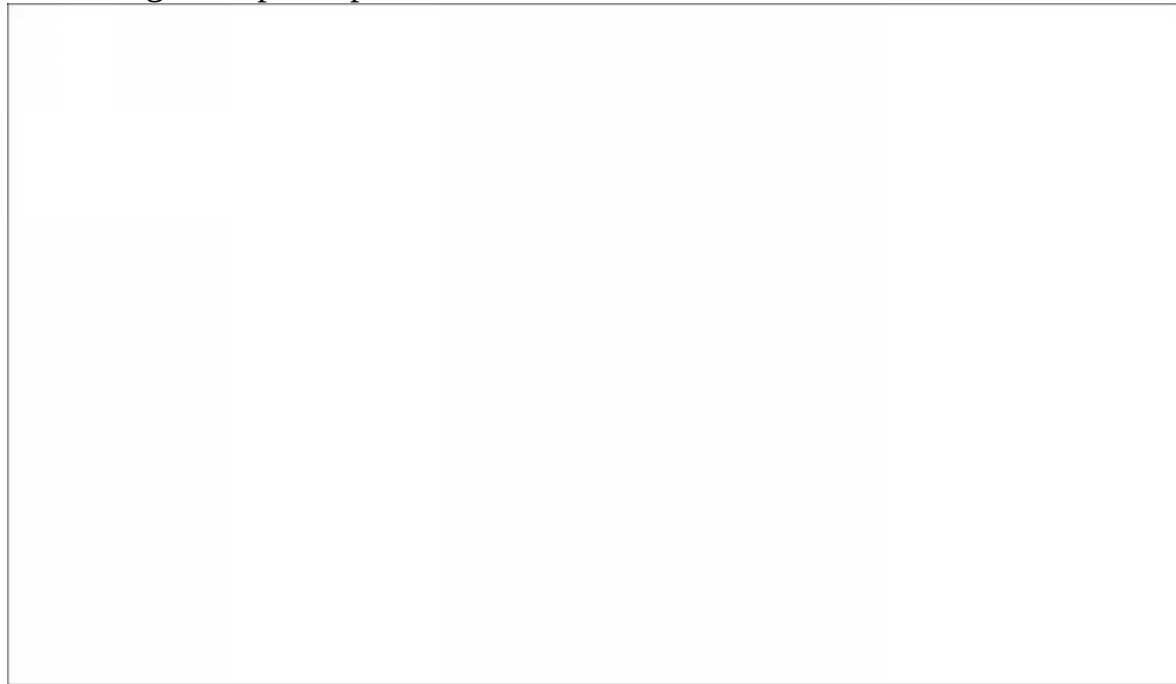
Using **const** keyword.

The #define Preprocessor

Following is the form to use #define preprocessor to define a constant:

`#define identifier value`

Following example explains it in detail:



```
#include <iostream>
```

```
using namespace std;
```

```
#define LENGTH 10
```

```
#define WIDTH 5
```

```
#define NEWLINE '\n'
```

```
int main()
```

```
{
```

```
    int area;
```

```
    area = LENGTH * WIDTH;
```


C++

```
        cout << area;  
  
        cout << NEWLINE;  
  
        return 0;  
  
    }
```

When the above code is compiled and executed, it produces the following result:

50

The const Keyword

You can use **const** prefix to declare constants with a specific type as follows:

```
const type variable = value;
```

Following example explains it in detail:

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    const int  LENGTH = 10;
```

```
    const int  WIDTH  = 5;
```

```
    const char NEWLINE = '\n';
```

```
    int area;
```

```
    area = LENGTH * WIDTH;
```

```
    cout << area;
```

```
    cout << NEWLINE;
```

```
        return 0;  
    }
```

When the above code is compiled and executed, it produces the following result:

50

Note that it is a good programming practice to define constants in CAPITALS.



C++

C++ allows the **char**, **int**, and **double** data types to have modifiers preceding them. A modifier is used to alter the meaning of the base type so that it more precisely fits the needs of various situations.

The data type modifiers are listed here:

signed

unsigned

long

short

The modifiers **signed**, **unsigned**, **long**, and **short** can be applied to integer base types. In addition, **signed** and **unsigned** can be applied to char, and **long** can be applied to double.

The modifiers **signed** and **unsigned** can also be

used

as

prefix

to **long** or **short** modifiers. For example, **unsigned long int**.

C++ allows a shorthand notation for declaring **unsigned**,

short, or **long** integers. You can simply use the word **unsigned**, **short**, or **long**, without **int**. It automatically implies **int**. For example, the following two statements both declare unsigned integer variables.

```
unsigned x;
```

```
unsigned int y;
```

To understand the difference between the way signed and unsigned integer modifiers are interpreted by C++, you should run the following short program:

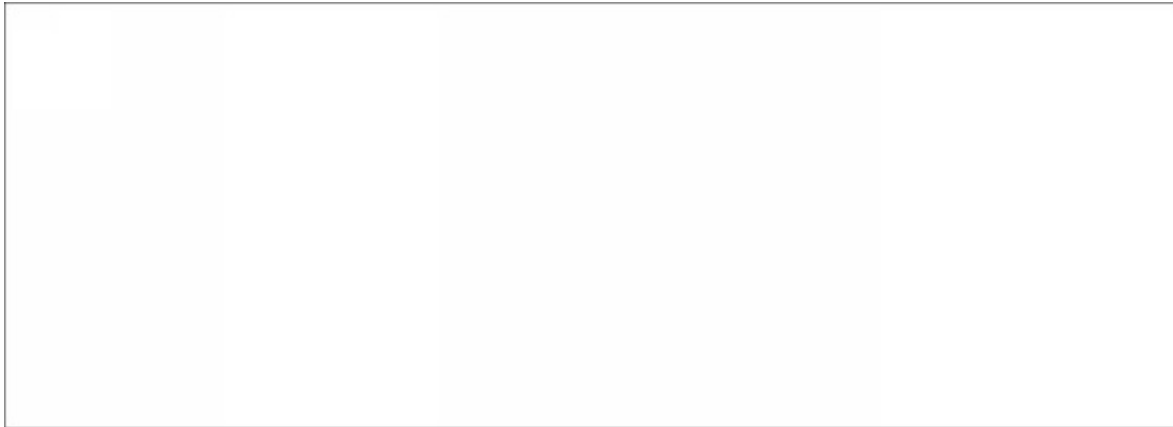
```
#include <iostream>
```

```
using namespace std;
```

```
/* This program shows the difference between  
signed and unsigned integers.
```

```
*/  
  
int main()  
{  
  
    short int i;  
    short unsigned int j;  
  
    // a signed short integer  
    // an unsigned short integer
```


C++



```
j = 50000;  
  
i = j;  
cout << i << " " << j;  
  
return 0;  
  
}
```

When this program is run, following is the output:



-15536 50000

The above result is because the bit pattern that represents 50,000 as a short unsigned integer is interpreted as -15,536 by a short.

Type Qualifiers in C++

The type qualifiers provide additional information about the variables they precede.

Qualifier Meaning

const Objects of type **const** cannot be changed by your program during execution

volatile The modifier **volatile** tells the compiler that a variable's value may be changed in ways not explicitly specified by the program.

restrict A pointer qualified by **restrict** is initially the only means by which the object it points to can be accessed. Only C99 adds a new type qualifier called restrict.



C++

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C++ Program. These specifiers precede the type that they modify. There are following storage classes, which can be used in a C++ Program

auto

register

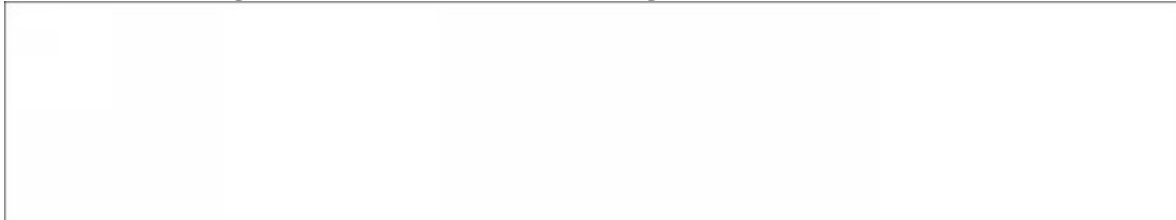
static

extern

mutable

The auto Storage Class

The **auto** storage class is the default storage class for all local variables.



{

```

        int mount;

        auto int month;

    }

```

The example above defines two variables with the same storage class, auto can only be used within functions, i.e., local variables.

The register Storage Class

The **register** storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).



```

{

        register int    miles;

}

```

The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.

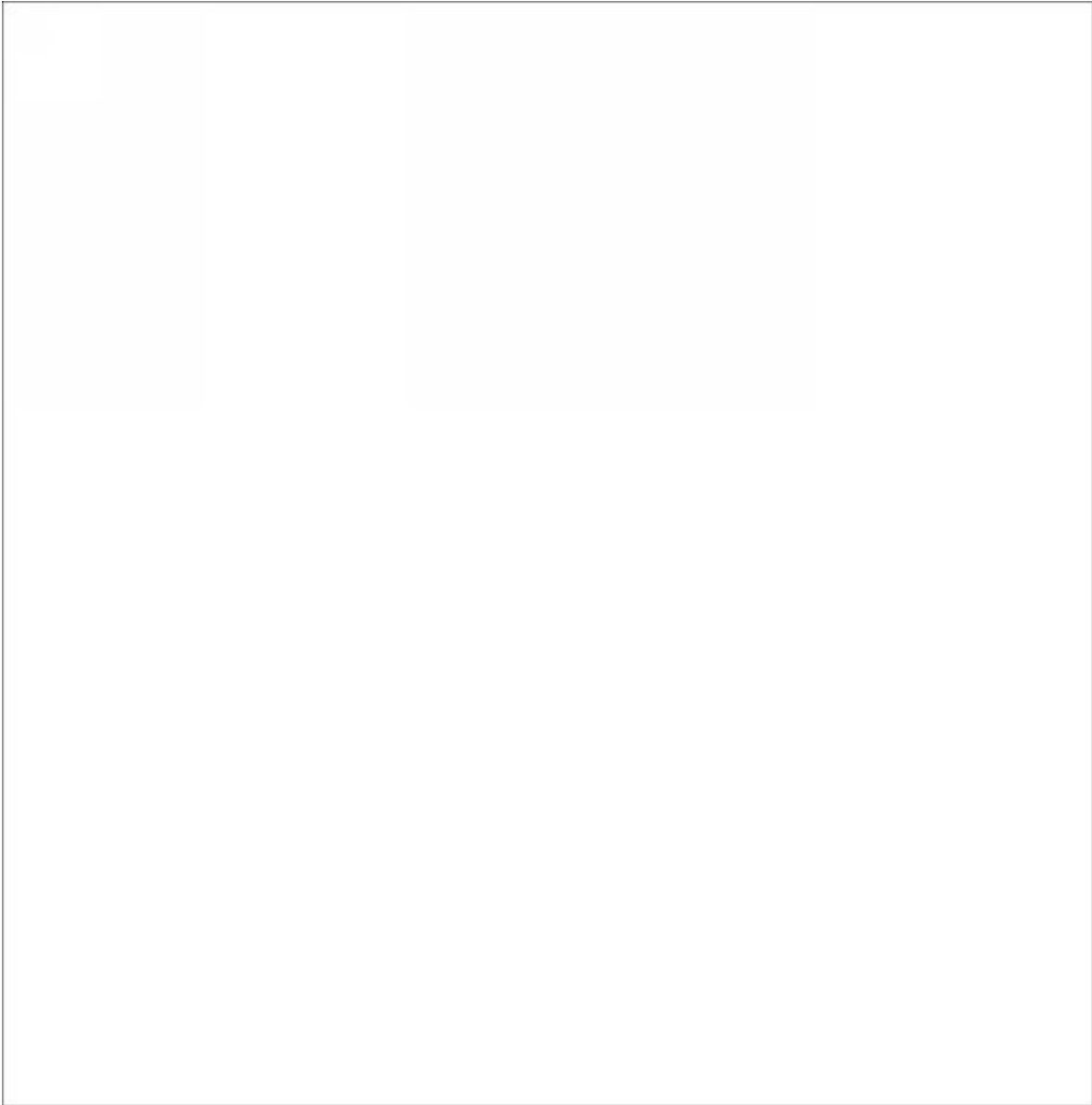
The static Storage Class

C++

The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

In C++, when static is used on a class data member, it causes only one copy of that member to be shared by all objects of its class.



```
#include <iostream>
```

```
Function declaration void func(void);
```

```
static int count = 10; /* Global variable */
```

```
main()
```

```
{
```

```

        while(count--)
        {
                                func();
        }
        return 0;
}

Function definition void func( void )
{

                                static int i = 5; // local static variable i++;

                                std::cout << "i is " << i ;

                                std::cout << " and count is " << count <<
                                std::endl;

}

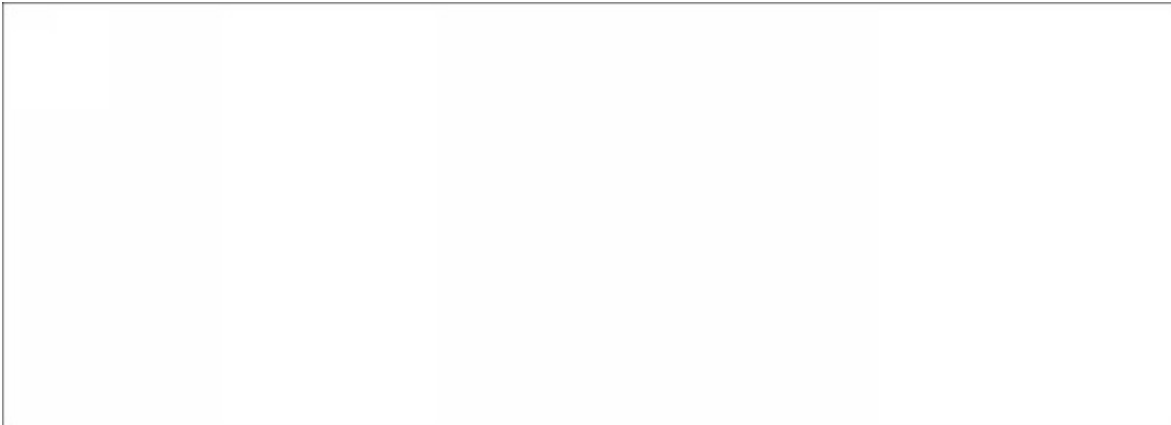
```

When the above code is compiled and executed, it produces the following result:

i is 6 and count is 9

i is 7 and count is 8

C++



i is 8 and count is 7

i is 9 and count is 6

i is 10 and count is 5

i is 11 and count is 4

i is 12 and count is 3

i is 13 and count is 2

i is 14 and count is 1

i is 15 and count is 0

The extern Storage Class

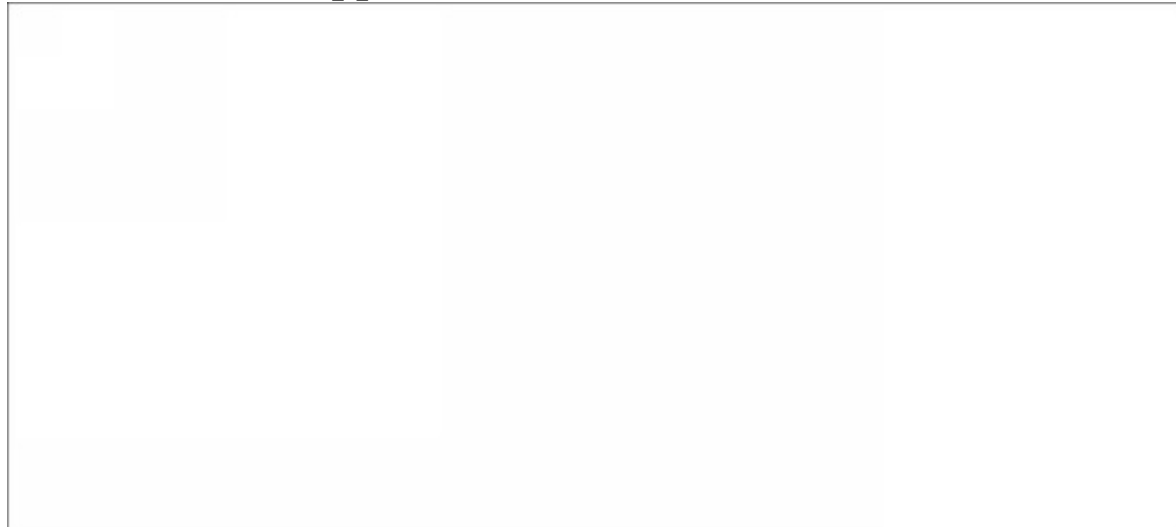
The **extern** storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern' the variable cannot be initialized as all it does is point the variable name at a storage location that has been previously defined.

When you have multiple files and you define a global variable or function, which will be used in other files also, then *extern* will be used in another file to give reference of defined variable or function. Just for understanding *extern* is used to declare a global variable or function in another file.

The extern modifier is most commonly used when there are two or more files

sharing the same global variables or functions as explained below.

First File: main.cpp



```
#include <iostream>

int count ;

extern void write_extern();

main()
{
    count = 5;
    write_extern();
}
```

Second File: support.cpp



```
#include <iostream>
```


C++

```
extern int count;

void write_extern(void)
{
    std::cout << "Count is " << count << std::endl;
}
```

Here, *extern* keyword is being used to declare count in another file. Now compile these two files as follows:

```
$g++ main.cpp support.cpp -o write
```

This will produce **write** executable program, try to execute **write** and check the result as follows:

```
$/write
```

```
5
```

The mutable Storage Class

The **mutable** specifier applies only to class objects, which are discussed later in this tutorial. It allows a member of an object to override const member function.

That is, a mutable member can be modified by a const member function.



C++

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C++ is rich in built-in operators and provide the following types of operators:

Arithmetic Operators

Relational Operators

Logical Operators

Bitwise Operators

Assignment Operators

Misc Operators

This chapter will examine the arithmetic, relational, logical, bitwise, assignment and other operators one by one.

Arithmetic Operators

There are following arithmetic operators supported by C++ language:

Assume variable A holds 10 and variable B holds 20, then:

Operator	Description			Example
+	Adds two operands			A + B will give 30
-	Subtracts second operand from			A - B will give -1
	the first			
*	Multiplies both operands			A * B will give 20
/	Divides numerator by de-			B / A will give 2
	numerator			
%	Modulus	Operator	and	B % A will give 0
	remainder of after an integer			
	division			

C++

++	Increment	operator,	increases	A++ will give 11
	integer value by one			
--	Decrement	operator,	decreases	A-- will give 9
	integer value by one			

Try the following example to understand all the arithmetic operators available in C++.

Copy and paste the following C++ program in test.cpp file and compile and run this program.

```
#include <iostream>
```

```
using namespace std;
```

```
main()
```

```
{
```

```
    int a = 21;
```


int b = 10;

int c ;

c = a + b;		
cout << "Line 1	- Value of c is :	<< c << endl ;
c = a - b;		
cout << "Line 2	- Value of c is	:< << c << endl ;
c = a * b;		
cout << "Line 3	- Value of c is :	<< c << endl ;
c = a / b;		
cout << "Line 4	- Value of c is	:< << c << endl ;
c = a % b;		
cout << "Line 5	- Value of c is	:< << c << endl ;
c = a++;		
cout << "Line 6	- Value of c is :	<< c << endl ;
c = a--;		
cout << "Line 7	- Value of c is	:< << c << endl ;
return 0;		

C++

}

When the above code is compiled and executed, it produces the following result:

- Line 1 - Value of c is :31
- Line 2 - Value of c is :11
- Line 3 - Value of c is :210
- Line 4 - Value of c is :2
- Line 5 - Value of c is :1
- Line 6 - Value of c is :21
- Line 7 - Value of c is :22

Relational Operators

There are following relational operators supported by C++ language

Assume variable A holds 10 and variable B holds 20, then:

Operator	Description				Exam

==	Checks	if	the	values	of	two	(A == is not true.
	operands are equal or not, if						
	yes then	condition			becomes		
	true.						
!=	Checks	if	the	values	of	two	(A != is true
	operands are equal or not, if						
	values are not equal then						
	condition becomes true.						
>	Checks	if	the	value	of	left	(A > B is not true.
	operand is greater than the						
	value of right operand, if yes						
	then condition becomes true.						
<	Checks	if	the	value	of	left	(A < B is true
	operand is less than the value						
	of right operand, if yes then						
	condition becomes true.						
>=	Checks	if	the	value	of	left	(A >= B is not true.



C++

	operand is greater than or equal				
	to the value of right operand, if				
	yes	then	condition	becomes	
	true.				
<=	Checks if		the value	of left	(A <= B) is true.
	operand is less than or equal to				
	the value of right operand, if				
	yes	then	condition	becomes	
	true.				

Try the following example to understand all the relational operators available in C++.

Copy and paste the following C++ program in test.cpp file and compile and run this program.

```
#include <iostream>
```

```
using namespace std;
```

```
main()
```

```
{
```

```
    int a = 21;
```

```
    int b = 10;
```

```
    int c ;
```

```
    if( a == b )
```

```
{  
  
    cout << "Line 1 - a is equal to  
    b" << endl ;  
  
}  
  
else  
  
{  
  
    cout << "Line 1 - a is not equal  
    to b" << endl ;  
  
}  
  
if ( a < b )  
  
{  
  
    cout << "Line 2 - a is less than  
    b" << endl ;
```

C++

}

else


```

{

    cout << "Line 2 - a is not less than
b" << endl ;

}

if ( a > b )

{

    cout << "Line 3 - a is greater than
b" << endl ;

}

else

{

    cout << "Line 3 - a is not greater
than b" << endl ;

}

/* Let's change the values of a and b */

a = 5;

b = 20;

if ( a <= b )

{

    cout << "Line 4 - a is either less
than \

        or equal to  b" << endl ;

}

if ( b >= a )

```

```
        {  
            cout << "Line 5 - b is either greater  
            than \  
            or equal to b" << endl ;  
        }  
    return 0;  
}
```

When the above code is compiled and executed, it produces the following result:



Line 1 - a is not equal to b

Line 2 - a is not less than b

Line 3 - a is greater than b

Line 4 - a is either less than or equal to b

Line 5 - b is either greater than or equal to b

Logical Operators

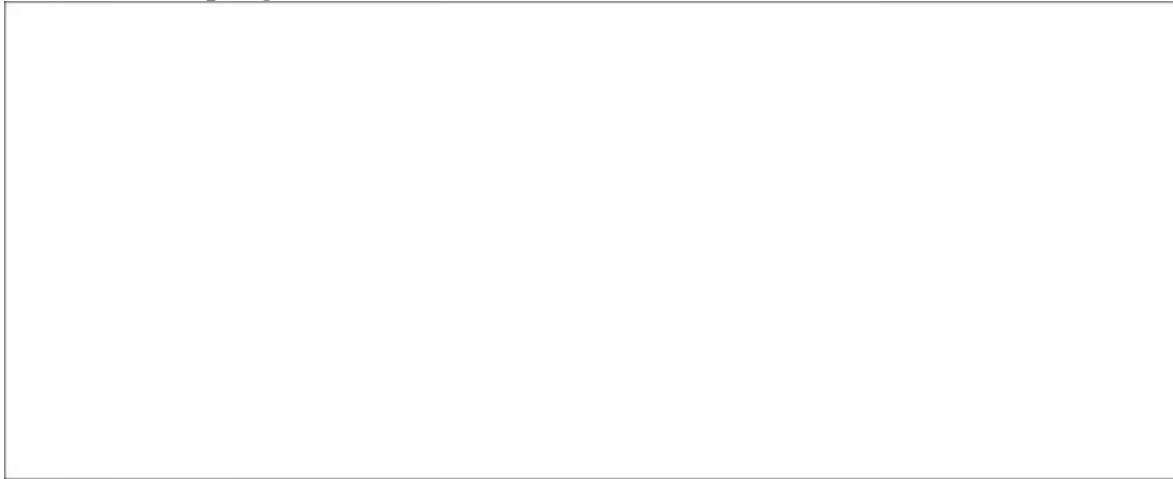
There are following logical operators supported by C++ language.

Assume variable A holds 1 and variable B holds 0, then:

Operator	Description		Example
&&	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.		(A && B) is false.
	Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.		(A B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.		!(A && B) is true.

Try the following example to understand all the logical operators available in C++.

Copy and paste the following C++ program in test.cpp file and compile and run this program.



```
#include <iostream>

using namespace std;

main()
{
    int a = 5;
    int b = 20;
    int c ;
```

C++

```
if ( a && b )
```

```
{
```

```
    cout << "Line 1 - Condition is true"  
    << endl ;
```

```
}  
  
if ( a || b )  
{  
  
    cout << "Line 2 - Condition is true"  
    << endl ;  
  
}  
  
/* Let's change the values of      a and b */  
  
a = 0;  
b = 10;  
  
if ( a && b )  
{  
  
    cout << "Line 3 - Condition is true"  
    << endl ;  
  
}  
  
else  
{  
  
    cout << "Line 4 - Condition is not  
    true"<< endl ;  
  
}  
  
if ( !(a && b) )  
{  
  
    cout << "Line 5 - Condition is true"  
    << endl ;  
  
}
```

```
        return 0;  
    }
```

When the above code is compiled and executed, it produces the following result:



Line 1 - Condition is true

Line 2 - Condition is true

Line 4 - Condition is not true

Line 5 - Condition is true

Bitwise Operators

C++

Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for $\&$, $|$, and \wedge are as follows:

p	q	p & q	p q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume if $A = 60$; and $B = 13$; now in binary format they will be as follows:

$A = 0011\ 1100$

$B = 0000\ 1101$

$A \& B = 0000\ 1100$

$A | B = 0011\ 1101$

$A \wedge B = 0011\ 0001$

$\sim A = 1100\ 0011$

The Bitwise operators supported by C++ language are listed in the following table. Assume variable A holds 60 and variable B holds 13, then:

--	--	--

Operator	Description	Example
&	Binary AND Operator copies a	(A & B) will give 12 which is
	bit to the result if it exists in	0000 1100
	both operands.	
	Binary OR Operator copies a bit	(A B) will give 61 which is
	if it exists in either operand.	0011 1101
^	Binary XOR Operator copies the	(A ^ B) will give 49 which is
	bit if it is set in one operand but	0011 0001
	not both.	
		42

~	Binary	Ones	Complement	(~A) will give -61 which is
	Operator is unary and has the			1100 0011 in 2's complement
	effect of 'flipping' bits.			form due to a signed binary
				number.
<<	Binary Left Shift Operator. The			A << 2 will give 240 which is
	left operands value is moved			1111 0000
	left by the number of bits			
	specified by the right operand.			
>>	Binary Right Shift Operator. The			A >> 2 will give 15 which is
	left operands value is moved			0000 1111
	right by the number of bits			
	specified by the right operand.			

Try the following example to understand all the bitwise operators available in C++.

Copy and paste the following C++ program in test.cpp file and compile and run this program.



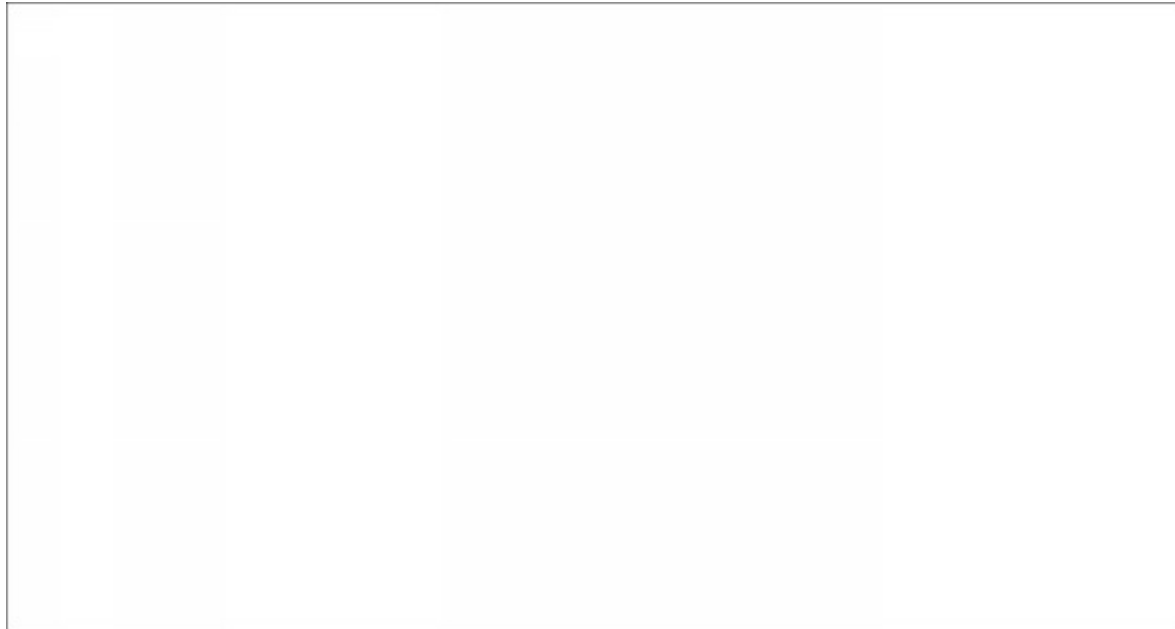
```
#include <iostream>
```

```
using namespace std;
```

main()	
{	
unsigned int a = 60;	// 60 = 0011 1100
unsigned int b = 13;	// 13 = 0000 1101
int c = 0;	
c = a & b;	// 12 = 0000 1100
cout << "Line 1 - Value of c is : " << c << endl ;	
c = a b;	// 61 = 0011 1101
cout << "Line 2 - Value of c is: " << c << endl ;	
c = a ^ b;	// 49 = 0011 0001

```
cout << "Line 3 - Value of c is: " << c << endl ;
```

C++



```
c = ~a; // -61 = 1100 0011
```

```
cout << "Line 4 - Value of c is: " << c << endl ;
```

```
c = a << 2; // 240 = 1111 0000
```

```
cout << "Line 5 - Value of c is: " << c << endl ;
```

```
c = a >> 2; // 15 = 0000 1111
```

```
cout << "Line 6 - Value of c is: " << c << endl ;
```

```
return 0;
```

```
}
```

When the above code is compiled and executed, it produces the following result:



Line 1 - Value of c is : 12

Line 2 - Value of c is: 61

Line 3 - Value of c is: 49

Line 4 - Value of c is: -61

Line 5 - Value of c is: 240

Line 6 - Value of c is: 15

Assignment Operators

There are following assignment operators supported by C++ language:

Operator	Description	Example
=	Simple assignment operator,	C = A + B will assign value of A
	Assigns values from right side	+ B into C
	operands to left side operand.	
+=	Add AND assignment operator,	C += A is equivalent to C = C +
	It adds right operand to the left	A

	operand and assign the result to	
	left operand.	
		44

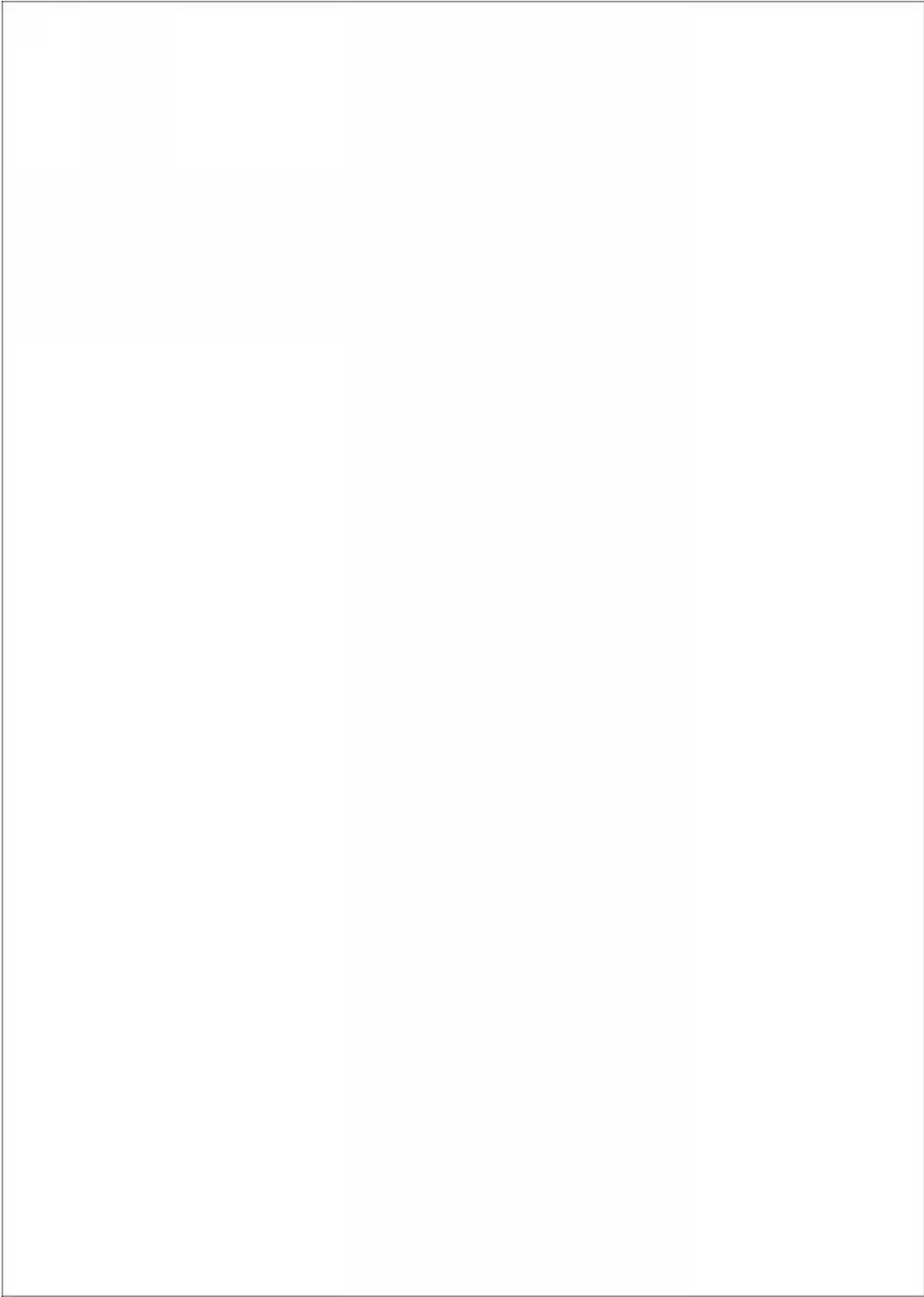
-=	Subtract	AND	assignment	C -= A is equivalent to	-
	operator,	It	subtracts	right	A
	operand from the left operand				
	and assign the result to left				
	operand.				
*=	Multiply	AND	assignment	C *= A is equivalent to	*
	operator,	It	multiplies	right	A
	operand with the left operand				
	and assign the result to left				
	operand.				
/=	Divide	AND	assignment	C /= A is equivalent to	/
	operator, It divides left operand				A
	with the right operand and				
	assign the result to left				
	operand.				
%=	Modulus	AND	assignment	C %= A is equivalent to	C
	operator,	It	takes modulus	% A	
	using two operands and assign				
	the result to left operand.				

<<=	Left	shift	AND	assignment	C <<= 2 is same as 2
	operator.				
>>=	Right	shift	AND	assignment	C >>= 2 is same as 2
	operator.				
&=	Bitwise	AND		assignment	C &= 2 is same as
	operator.				
^=	Bitwise	exclusive	OR	and	C ^= 2 is same as C
	assignment operator.				
=	Bitwise	inclusive	OR	and	C = 2 is same as C
	assignment operator.				

Try the following example to understand all the assignment operators available in C++.

C++

Copy and paste the following C++ program in test.cpp file and compile and run this program.



#include <iostream>

```
using namespace std;
```

```
main()
```

```
{
```

```
    int a = 21;
```

```
    int c ;
```

```
    c =  a;
```

```
    cout << "Line 1 - =      Operator, Value of c = : " <<c<<endl ;
```

```
    c +=  a;
```

```
    cout << "Line 2 - += Operator, Value of c = : " <<c<<endl ;
```

```
    c -=  a;
```

```
    cout << "Line 3 - -= Operator, Value of c = : " <<c<<endl ;
```

```
    c *=  a;
```

```
    cout << "Line 4 - *= Operator, Value of c = : " <<c<<endl ;
```

```
    c /=  a;
```

```
    cout << "Line 5 - /= Operator, Value of c = : " <<c<<endl ;
```

```
    0= 200;
```

```
c %= a;
```

```
cout << "Line 6 - %= Operator, Value of c = : " <<c<<  
endl ;
```

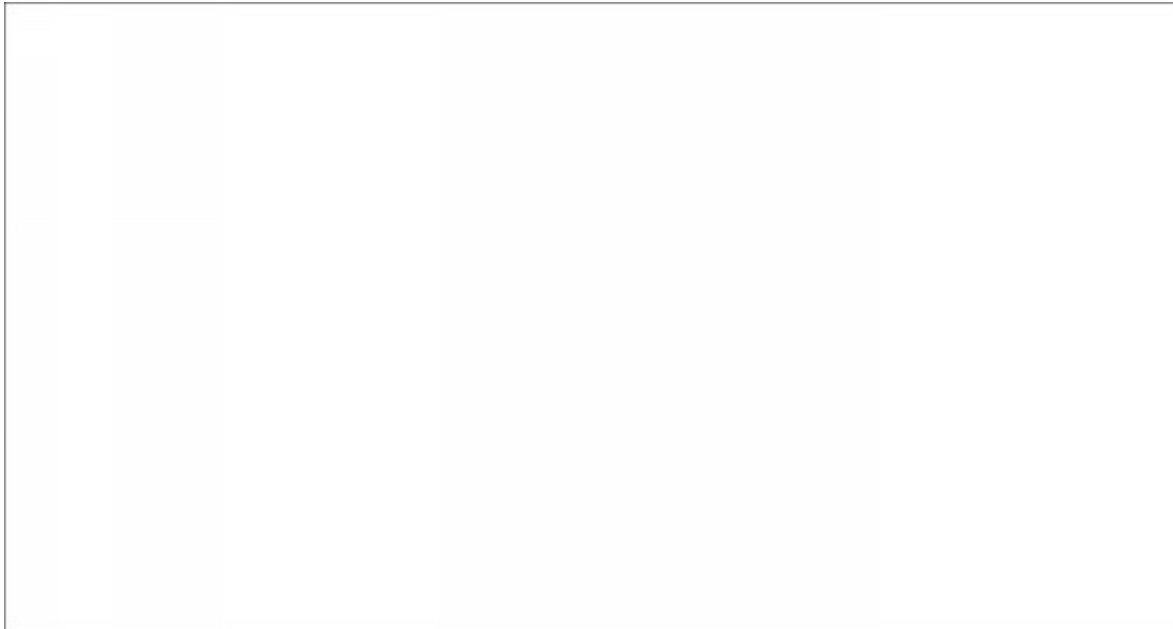
```
c <<= 2;
```

```
cout << "Line 7 - <<= Operator, Value of c = : " <<c<<  
endl ;
```

```
c >>= 2;
```

```
cout << "Line 8 - >>= Operator, Value of c = : " <<c<<  
endl ;
```

C++



```
c &= 2;
```

```
cout << "Line 9 - &= Operator, Value of c = : "  
<<c<< endl ;
```

```
c ^= 2;
```

```
cout << "Line 10 - ^= Operator, Value of c = : "  
<<c<< endl ;
```

```
c |= 2;
```

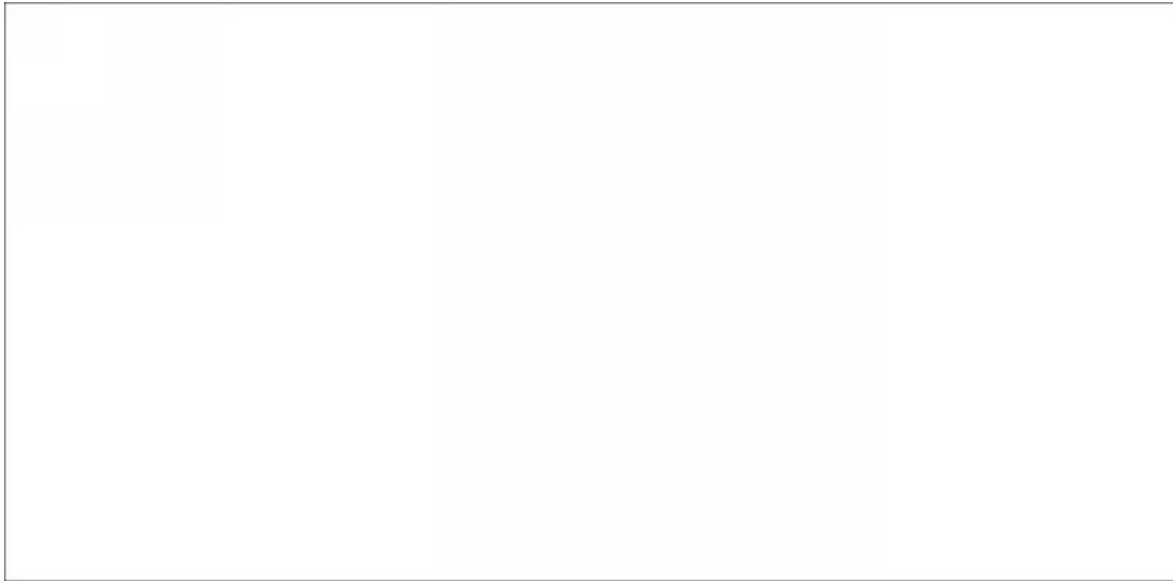
```
cout << "Line 11 - |= Operator, Value of c = : "  
<<c<< endl ;
```

```
return 0;
```

```
}
```

When the above code is compiled and executed, it produces the following

result:



Line 1 - = Operator, Value of c = : 21

Line 2 - += Operator, Value of c = : 42

Line 3 - -= Operator, Value of c = : 21

Line 4 - *= Operator, Value of c = : 441

Line 5 - /= Operator, Value of c = : 21

Line 6 - %= Operator, Value of c = : 11

Line 7 - <<= Operator, Value of c = : 44

Line 8 - >>= Operator, Value of c = : 11

Line 9 - &= Operator, Value of c = : 2

Line 10 - ^= Operator, Value of c = : 0

Line 11 - |= Operator, Value of c = : 2

Misc Operators

The following table lists some other operators that C++ supports.



Operator	Description
sizeof	sizeof operator returns the size of a variable. For example, sizeof(a), where 'a' is integer, and will return
	4.
	47

	C++
Condition ? X : Y	Conditional operator (?). If Condition is true then it returns value of X otherwise returns value of Y.
,	Comma operator causes a sequence of operations to be performed. The value of the entire comma expression is the value of the last expression of the comma-separated list.
. (dot) and -> (arrow)	Member operators are used to reference individual members of classes, structures, and unions.
Cast	Casting operators convert one data type to another. For example, int(2.2000) would return 2.
&	Pointer operator '&' returns the address of a variable. For example &a; will give actual address of the variable.
*	Pointer operator * is pointer to a variable. For example *var; will pointer to a variable var.

Operators Precedence in C++

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

For example $x = 7 + 3 * 2$; here, x is assigned 13, not 20 because operator $*$ has higher precedence than $+$, so it first gets multiplied with $3*2$ and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Postfix	() [] -> . ++ --	Left to right

Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
		48

		C++
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR		Left to right
Logical AND	&&	Left to right
Logical OR		Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= =	Right to left
Comma	,	Left to right

Try the following example to understand operator's precedence concept

available in C++. Copy and paste the following C++ program in test.cpp file and compile and run this program.

Check the simple difference with and without parenthesis. This will produce different results because (), /, * and + have different precedence. Higher precedence operators will be evaluated first:

```
#include <iostream>

using namespace std;

main()
{
```

```
    int a = 20;
```

C++



```
int b = 10;

int c = 15;

int d = 5;

int e;
```

e = (a + b) * c /	//	(30 *) / 5
d;		15	
cout << "Value of (a + b) * c / d is : " << e << endl ;			
e = ((a + b) * c)	//	(30 * 15) / 5	
/ d;			

cout << "Value of ((a + b) *	c) / d	:" << e <<
	is	endl ;

e = (a + b) * (c /	//	(30) * (15/5)
d);		
cout << "Value of (a + b) *	(c / d) is	:"
		<<
		e
		<<
		endl
		;
e = a + (b * c) /	//	20
d;		+ (150/5)
cout << "Value	* c) /	d
of a + (b		is
		:"
		<<
		e
		<<
		endl
		;

return 0;

}

When the above code is compiled and executed, it produces the following result:

Value of (a + b) * c / d is :90

Value of ((a + b) * c) / d is :90

Value of (a + b) * (c / d) is :90

Value of $a + (b * c) / d$ is :50

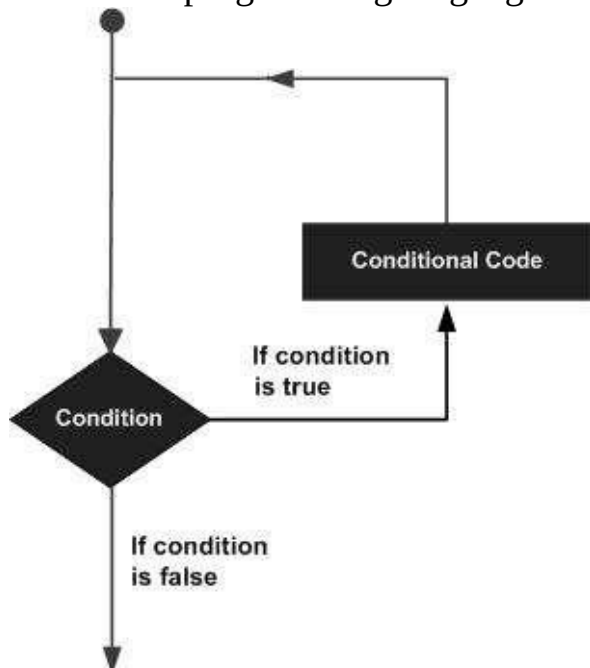


C++

There may be a situation, when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general from of a loop statement in most of the programming languages:



C++ programming language provides the following type of loops to handle looping requirements.

Loop Type	Description
while loop	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
for loop	Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
	51

	C++
do...while loop	Like a 'while' statement, except that it tests the condition at the end of the loop body.
nested loops	You can use one or more loop inside any another 'while', 'for' or 'do..while' loop.

While Loop

A **while** loop statement repeatedly executes a target statement as long as a given condition is true.

Syntax

The syntax of a while loop in C++ is:

```
while(condition)
{
    statement(s);
}
```

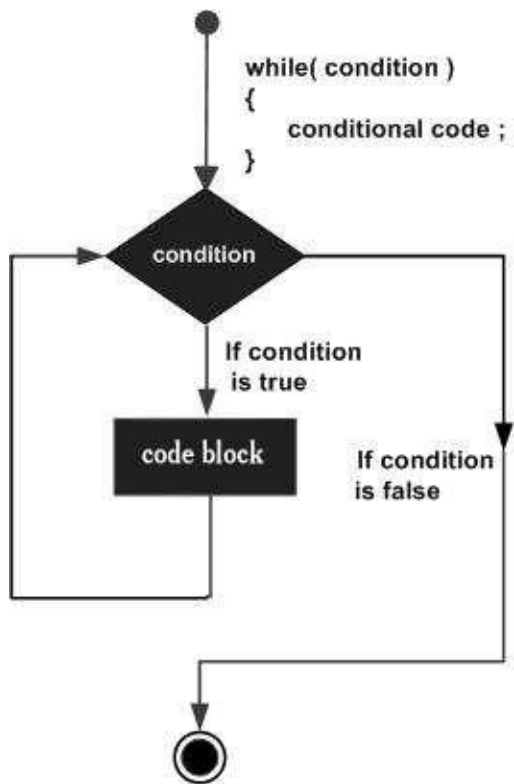
Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any non-zero value. The loop iterates while the condition is true.

When the condition becomes false, program control passes to the line

immediately following the loop.

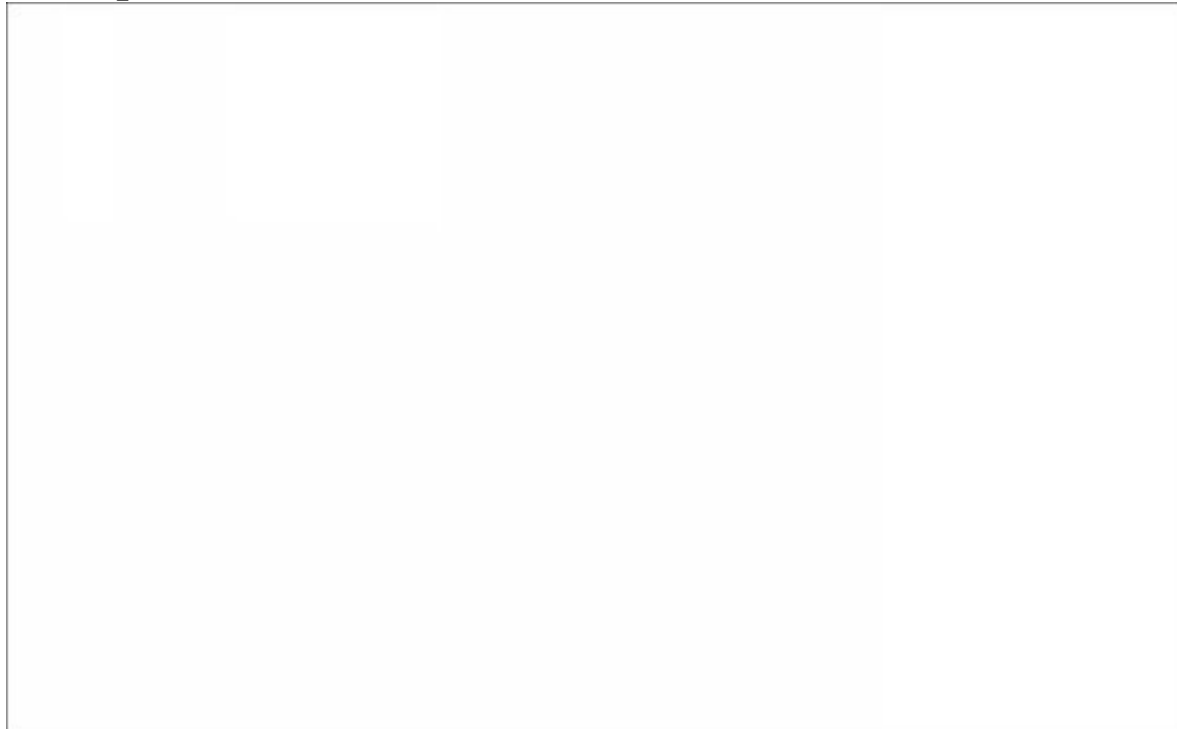
Flow Diagram

C++



Here, key point of the *while* loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example



```
#include <iostream>
```

```
using namespace std;
```

```
int main ()
```

```
{
```

Local variable declaration: int a = 10;

while loop execution while(a < 20)

```
{
```

```
cout << "value of a: " <<  
a << endl; a++;
```

```
}
```

C++

```
        return 0;  
    }
```

When the above code is compiled and executed, it produces the following result:

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 15

value of a: 16

value of a: 17

value of a: 18

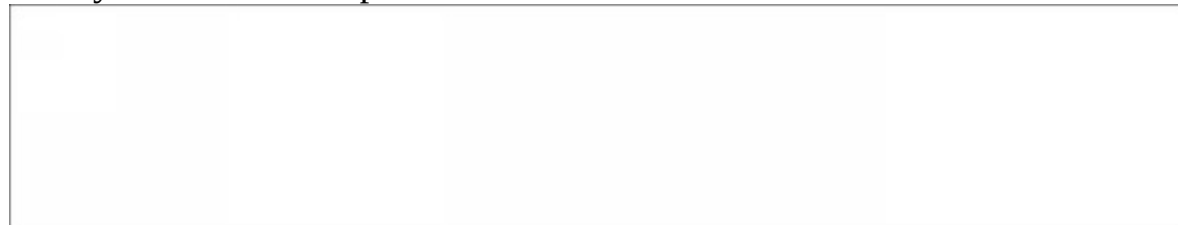
value of a: 19

for Loop

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax

The syntax of a for loop in C++ is:



```
for ( init; condition; increment )  
{  
  
    statement(s);  
  
}
```

Here is the flow of control in a for loop:

The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.

Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.

After the body of the for loop executes, the flow of control jumps back up to the **increment**

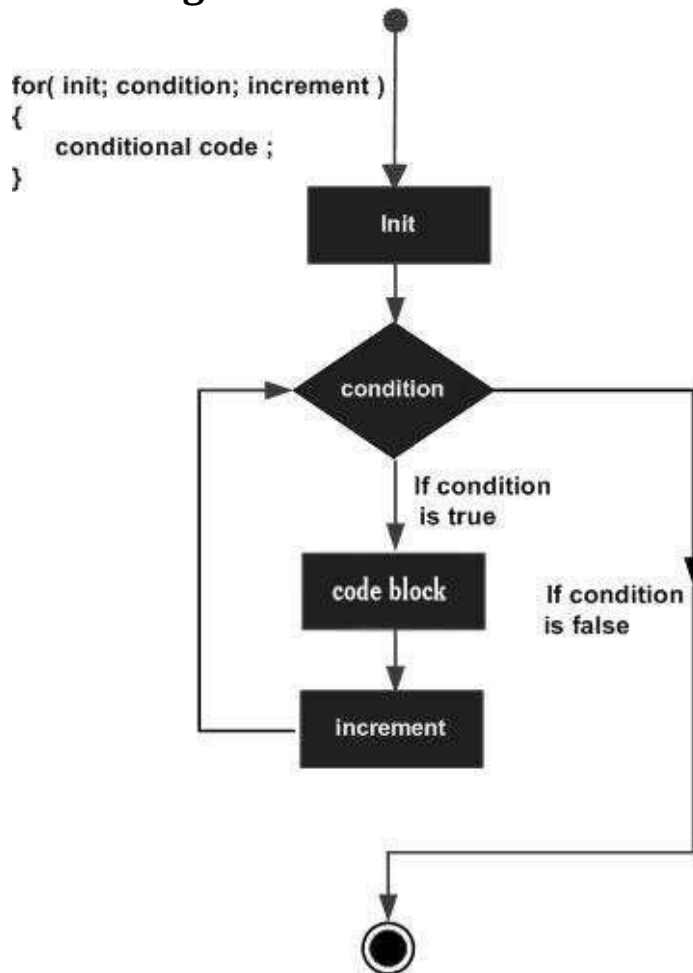
statement. This statement allows you to update any

C++

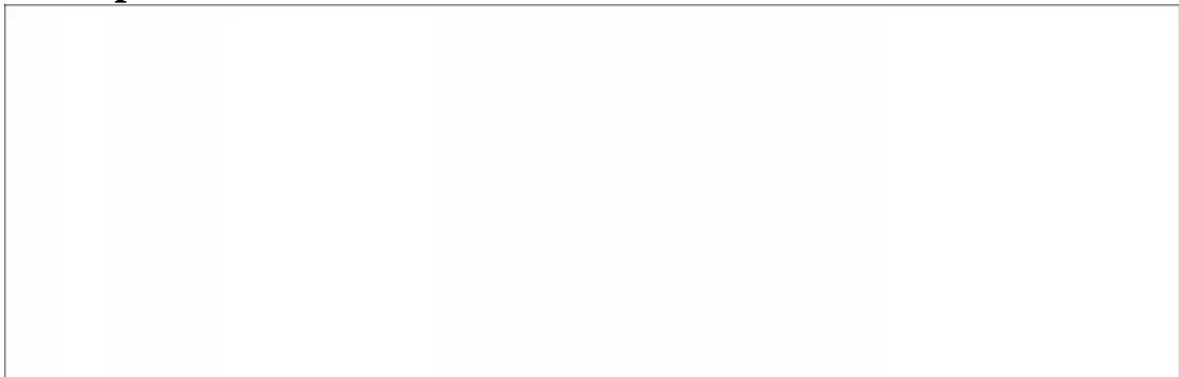
loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.

The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the for loop terminates.

Flow Diagram



Example



```
#include <iostream>
```

```
using namespace std;
```

```
int main ()
```

```
{
```

```
    // for loop execution
```

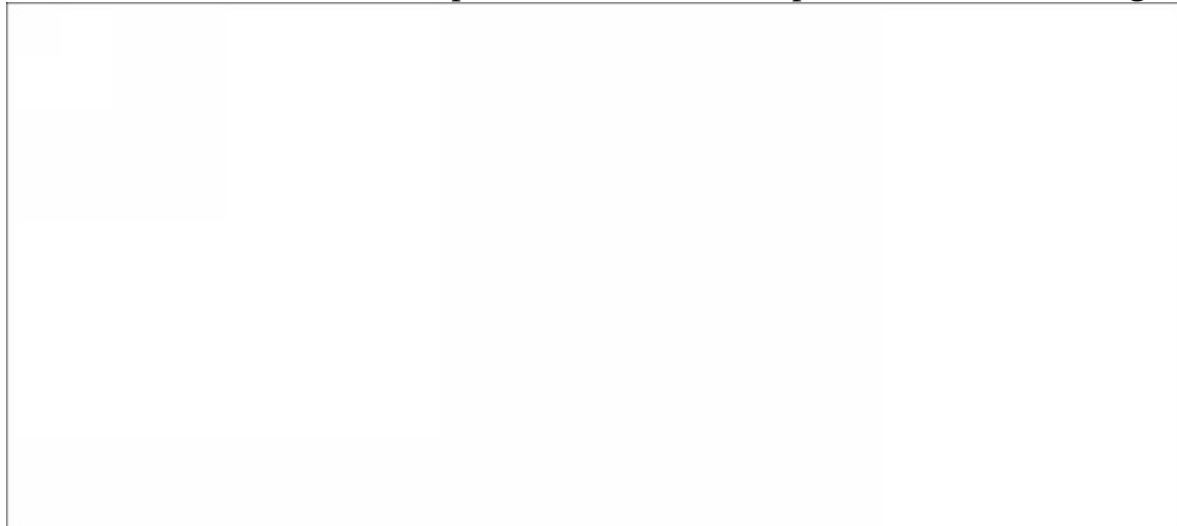
```
    for( int a = 10; a < 20; a = a + 1 )
```

C++



```
{  
  
    cout << "value of a: " <<  
    a << endl;  
  
}  
  
return 0;  
  
}
```

When the above code is compiled and executed, it produces the following result:



value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 15

value of a: 16

value of a: 17

value of a: 18

value of a: 19

do...while Loop

Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do...while** loop checks its condition at the bottom of the loop.

A **do...while** loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

Syntax

The syntax of a do...while loop in C++ is:

do

{

statement(s);

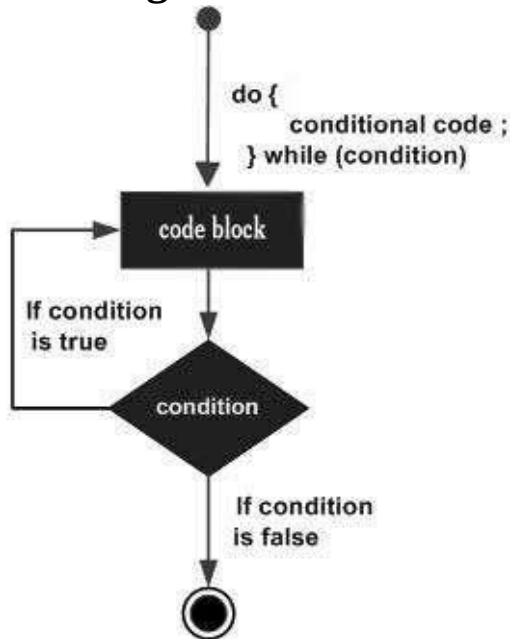
}while(condition);

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop execute once before the condition is tested.

C++

If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop execute again. This process repeats until the given condition becomes false.

Flow Diagram



Example

```
#include <iostream>

using namespace std;
```

```
int main ()
{
```

23 Local variable declaration: int a = 10;

24 do loop execution

```
do
{
```

```
        cout << "value of a: " <<  
        a << endl;  
  
        a = a + 1;  
  
    }while( a < 20 );  
  
    return 0;
```

C++

```
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10
```

```
value of a: 11
```

```
value of a: 12
```

```
value of a: 13
```

```
value of a: 14
```

```
value of a: 15
```

```
value of a: 16
```

```
value of a: 17
```

```
value of a: 18
```

```
value of a: 19
```

nested Loops

A loop can be nested inside of another loop. C++ allows at least 256 levels of nesting.

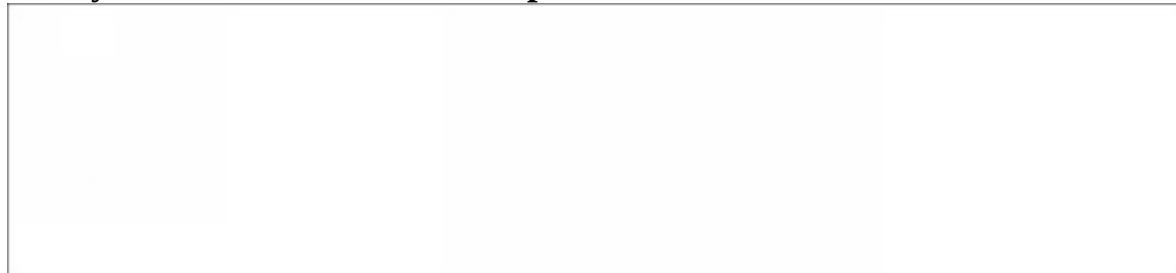
Syntax

The syntax for a **nested for loop** statement in C++ is as follows:



```
for ( init; condition; increment )
{
    for ( init; condition; increment )
    {
        statement(s);
    }
    statement(s); // you can put more statements.
}
```

The syntax for a **nested while loop** statement in C++ is as follows:



```
while(condition)
```

```
{
```

```
    while(condition)
```

```
    {
```

```
        statement(s);
```

C++

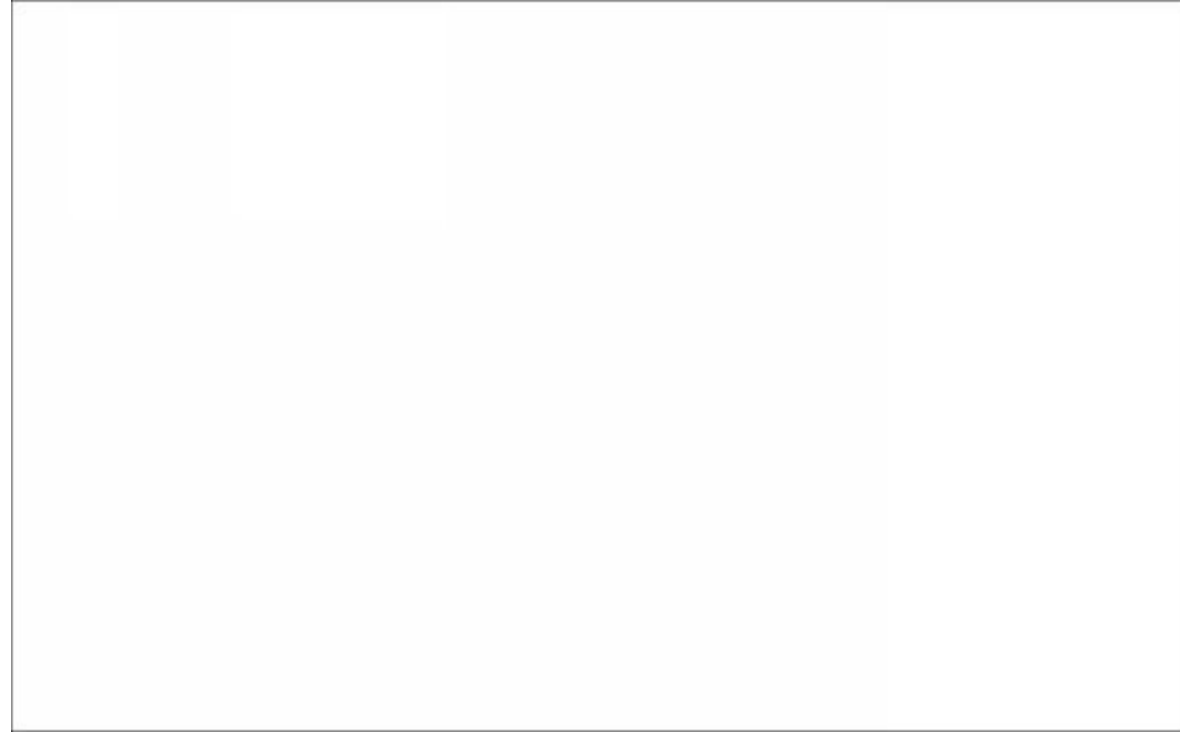
```
    }  
    statement(s); // you can put more statements.  
}
```

The syntax for a **nested do...while loop** statement in C++ is as follows:

```
do  
{  
    statement(s); // you can put more statements.  
    do  
    {  
        statement(s);  
    }while( condition );  
}while( condition );
```

Example

The following program uses a nested for loop to find the prime numbers from 2 to 100:



```
#include <iostream>
```

```
using namespace std;
```

```
int main ()
```

```
{
```

```
    int i, j;
```

```
    for(i=2; i<100; i++) {
```

```
        for(j=2; j <= (i/j); j++)
```

```
            if(!(i%j)) break; // if  
            factor found, not prime
```



```
        if(j > (i/j)) cout << i << "  
        is prime\n";  
    }  
    return 0;  
}
```

This would produce the following result:

C++

2 is prime

3 is prime

5 is prime

7 is prime

11 is prime

13 is prime

17 is prime

19 is prime

23 is prime

29 is prime

31 is prime

37 is prime

41 is prime

43 is prime

47 is prime

53 is prime

59 is prime

61 is prime

67 is prime

71 is prime

73 is prime

79 is prime

83 is prime

89 is prime

97 is prime

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

C++ supports the following control statements.

Control Statement	Description		
break statement	Terminates	the loop or switch statement	and
	transfers execution to the statement immediately		
			60

	C++
	following the loop or switch.
continue statement	Causes the loop to skip the remainder of its body and
	immediately retest its condition prior to reiterating.
goto statement	Transfers control to the labeled statement. Though it
	is not advised to use goto statement in your program.

Break Statement

The **break** statement has the following two usages in C++:

When the **break** statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.

It can be used to terminate a case in the **switch** statement (covered in the next chapter).

If you are using nested loops (i.e., one loop inside another loop), the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

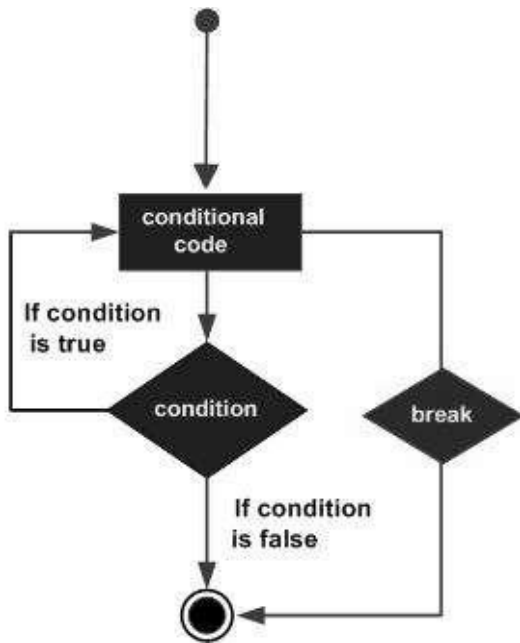
Syntax

The syntax of a break statement in C++ is:

```
break;
```

Flow Diagram

C++



Example



```
#include <iostream>
```

```
using namespace std;
```

```
int main ()
```

```
{
```

```
    23 Local variable declaration: int a = 10;
```

```
    24 do loop execution
```

```
do
```



```
{  
  
    cout << "value of a: " <<  
    a << endl;  
  
    a = a + 1;  
  
    if( a > 15)  
    {  
  
        terminate the loop  
        break;  
    }  
  
}while( a < 20 );
```

C++

```
        return 0;  
    }
```

When the above code is compiled and executed, it produces the following result:

```
value of a: prettyprint notranslate10  
value of a: 11  
value of a: 12  
value of a: 13  
value of a: 14  
value of a: 15
```

continue Statement

The **continue** statement works somewhat like the break statement. Instead of forcing termination, however, continue forces the next iteration of the loop to take place, skipping any code in between.

For the **for** loop, continue causes the conditional test and increment portions of the loop to execute. For the **while** and **do...while** loops, program control passes to the conditional tests.

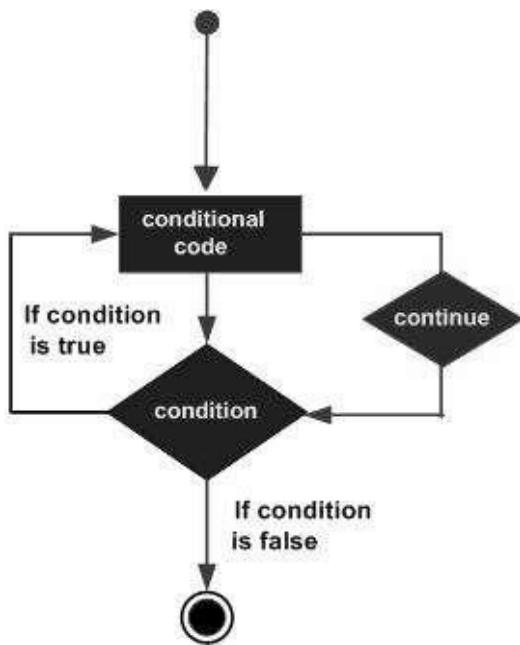
Syntax

The syntax of a continue statement in C++ is:

```
continue;
```

Flow Diagram

C++



Example



```
#include <iostream>
```

```
using namespace std;
```

```
int main ()
```

```
{
```

```
    Local variable declaration: int a = 10;
```

```
    do loop execution
```

```
do
```

```
{
```

```
    if( a == 15)
```

```
    {
```

```
        skip the iteration. a = a  
        + 1; continue;
```

```
    }
```

```
    cout << "value of a: " <<  
    a << endl;
```

```
    a = a + 1;
```

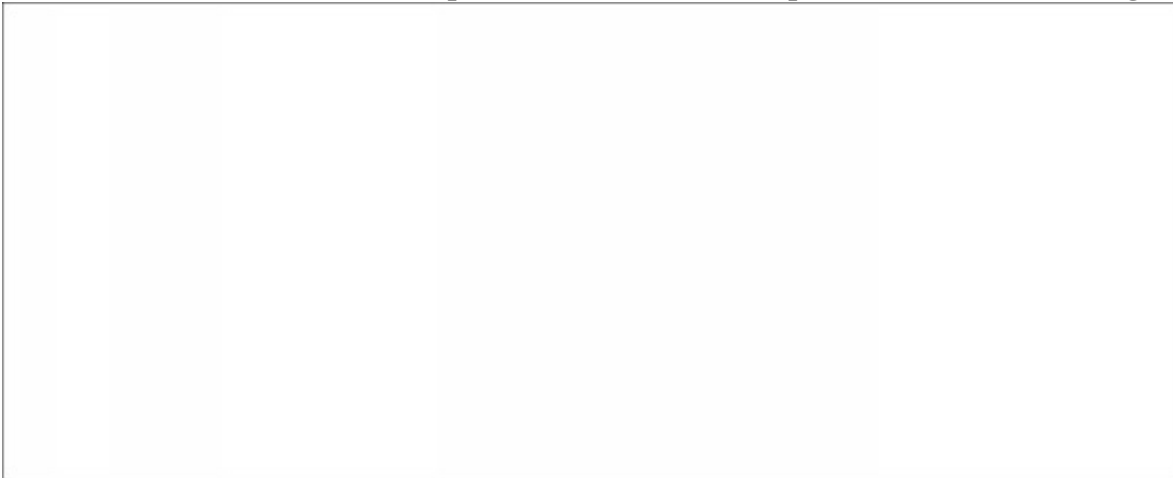
```
}while( a < 20 );
```

C++



```
        return 0;  
    }
```

When the above code is compiled and executed, it produces the following result:



value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 16

value of a: 17

value of a: 18

value of a: 19

goto Statement

A **goto** statement provides an unconditional jump from the goto to a labeled statement in the same function.

NOTE: Use of **goto** statement is highly discouraged because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a goto can be rewritten so that it doesn't need the goto.

Syntax

The syntax of a goto statement in C++ is:



```
goto label;
```

```
..
```

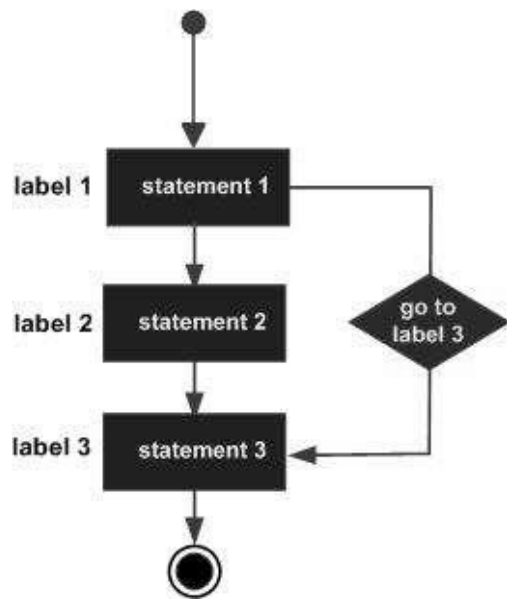
```
.
```

```
label: statement;
```

Where **label** is an identifier that identifies a labeled statement. A labeled statement is any statement that is preceded by an identifier followed by a colon (:).

Flow Diagram

C++



Example



```
#include <iostream>
```

```
using namespace std;
```

```
int main ()
```

```
{
```

```
    Local variable declaration: int a = 10;
```

```
    do loop execution
```

```
LOOP:do
```

```
{  
  
    if( a == 15)  
    {  
        skip the iteration. a = a  
        + 1;  
  
        goto LOOP;  
    }  
  
    cout << "value of a: " <<  
    a << endl;  
  
    a = a + 1;  
  
}while( a < 20 );
```

C++

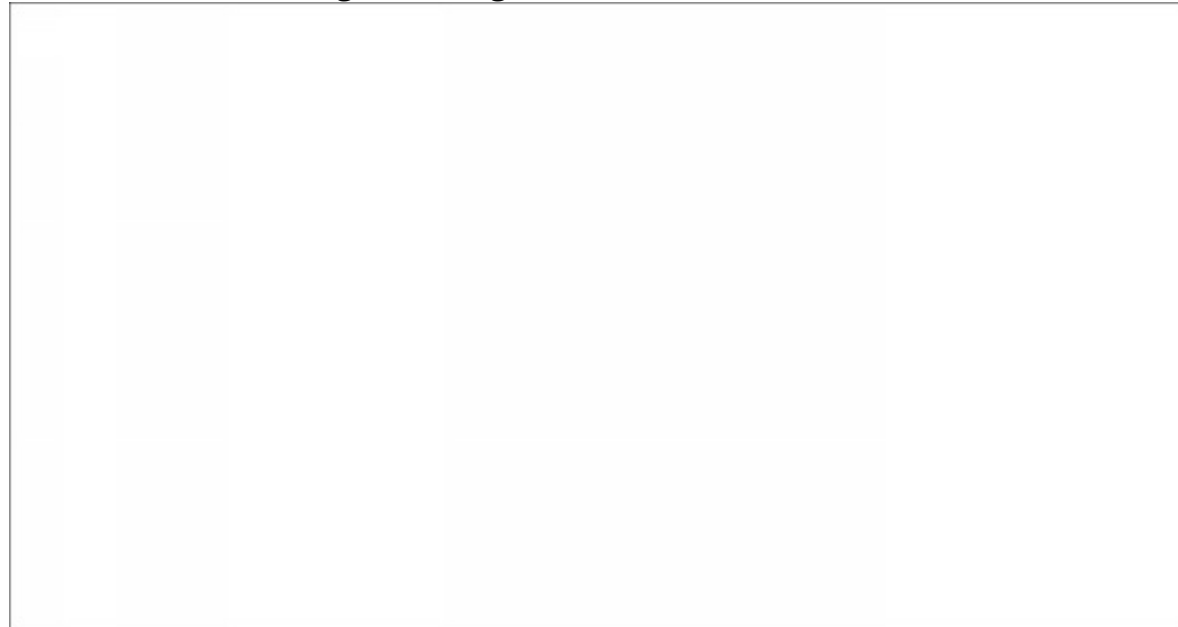
```
        return 0;  
    }
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10  
value of a: 11  
value of a: 12  
value of a: 13  
value of a: 14  
value of a: 16  
value of a: 17  
value of a: 18  
value of a: 19
```

One good use of goto is to exit from a deeply nested routine. For example,

consider the following code fragment:



```
for(...) {  
    for(...) {  
        while(...) {  
            if(...) goto stop;  
            .  
            .  
            .  
        }  
    }  
}  
  
stop:  
    cout << "Error in program.\n";
```

Eliminating the **goto** would force a number of additional tests to be performed. A simple **break** statement would not work here, because it would only cause the

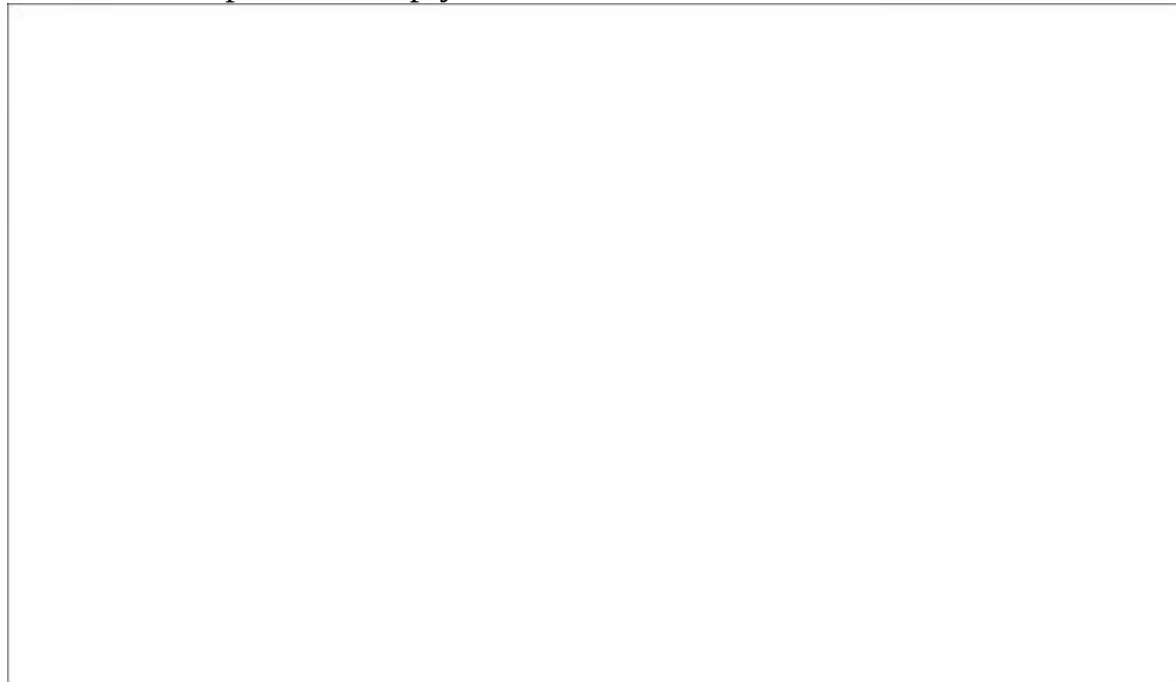
program to exit from the innermost loop.

The Infinite Loop

A loop becomes infinite loop if a condition never becomes false. The **for** loop is traditionally used for this purpose. Since none of the three expressions that form

C++

the 'for' loop are required, you can make an endless loop by leaving the conditional expression empty.



```
#include <iostream>
```

```
using namespace std;
```

```
int main ()
```

```
{
```

```
    for( ; ; )
```

```
    {
```

```
        printf("This loop will run  
        forever.\n");
```

```
    }
```

```
        return 0;  
  
    }
```

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C++ programmers more commonly use the ‘for (;;)’ construct to signify an infinite loop.

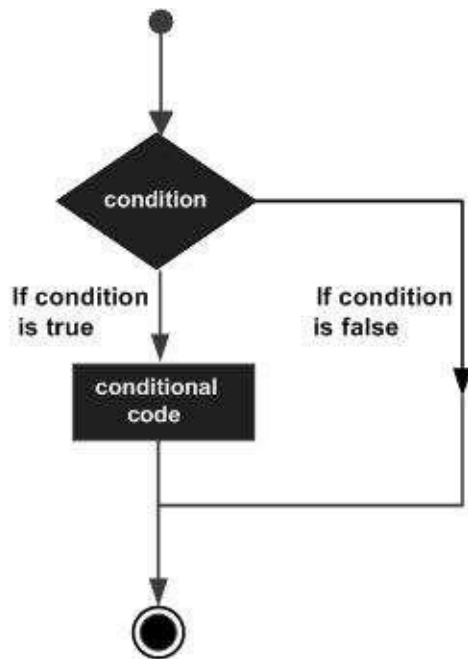
NOTE: You can terminate an infinite loop by pressing Ctrl + C keys.



C++

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical decision making structure found in most of the programming languages:



C++ programming language provides following types of decision making statements.

Statement	Description

if statement	An 'if' statement consists of a boolean expression followed by one or more statements.
if...else statement	An 'if' statement can be followed by an optional 'else' statement, which executes when the boolean expression is false.
switch statement	A 'switch' statement allows a variable to be tested
	69

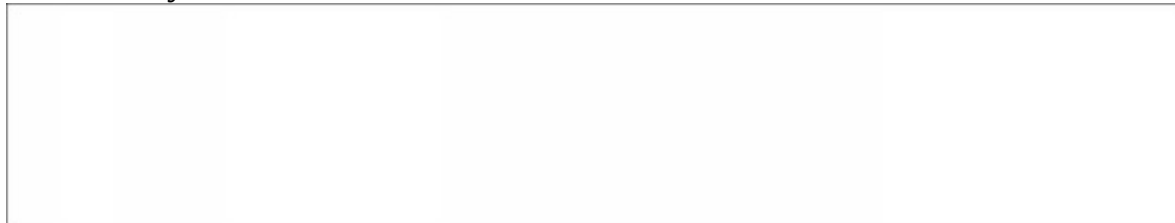
	C++
	for equality against a list of values.
nested if statements	You can use one 'if' or 'else if' statement inside another 'if' or 'else if' statement(s).
nested switch statements	You can use one 'switch' statement inside another 'switch' statement(s).

If Statement

An **if** statement consists of a boolean expression followed by one or more statements.

Syntax

The syntax of an if statement in C++ is:



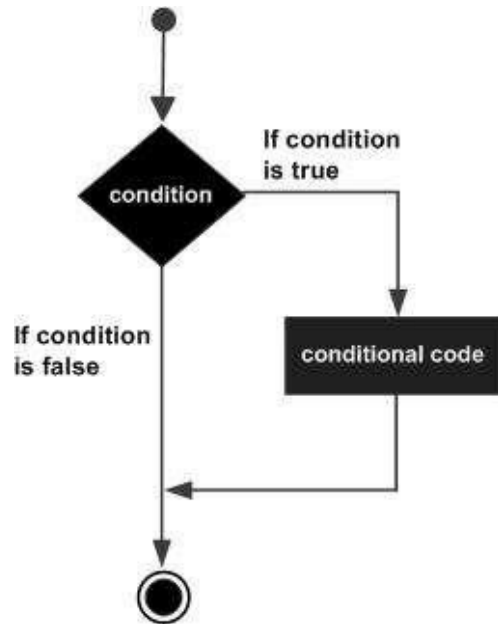
```
if(boolean_expression)
{
    // statement(s) will execute if the boolean
    // expression is true
}
```

If the boolean expression evaluates to **true**, then the block of code inside the if statement will be executed. If boolean expression evaluates to **false**, then the first set of code after the end of the if statement (after the closing curly brace) will be executed.

Flow Diagram

70

C++



Example

```
#include <iostream>
```

```
using namespace std;
```

```
int main ()
```

```
{
```

```
    local variable declaration: int a = 10;
```

```
    check the boolean condition if( a < 20 )
```

```
{
```

```

        if condition is true
        then print the following
        cout << "a is less than
        20;" << endl;

    }

    cout << "value of a is : " << a << endl;

    return 0;

}

```

C++

When the above code is compiled and executed, it produces the following result:

a is less than 20;

value of a is : 10

if...else Statement

An **if** statement can be followed by an optional **else** statement, which executes when the boolean expression is false.

Syntax

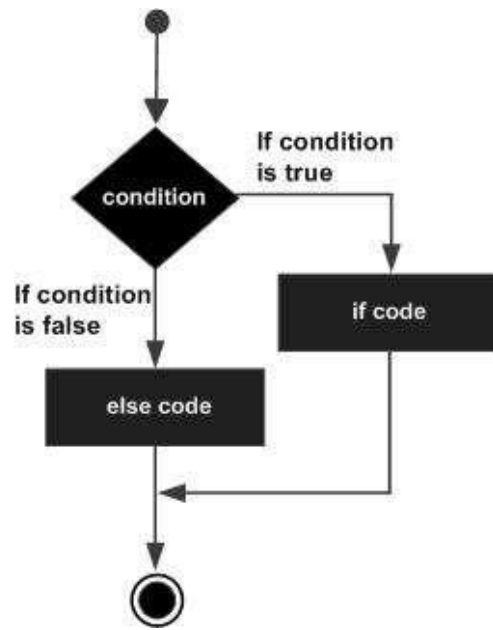
The syntax of an if...else statement in C++ is:



```
if(boolean_expression)
{
    // statement(s) will execute if the boolean expression is
    true
}
else
{
    // statement(s) will execute if the boolean expression is false
}
```

If the boolean expression evaluates to **true**, then the **if block** of code will be executed, otherwise **else block** of code will be executed.

Flow Diagram



C++

Example

```
#include <iostream>

using namespace std;

int main ()
{
```

```

local variable declaration: int a = 100;

check the boolean condition if( a < 20 )

{
    if condition is true
    then print the following

    cout << "a is less than
    20;" << endl;

}
else
{
    if condition is false
    then print the following
    cout << "a is not less than
    20;" << endl;

}

cout << "value of a is : " << a << endl;

return 0;

}

```

When the above code is compiled and executed, it produces the following result:

a is not less than 20;

value of a is : 100

if...else if...else Statement

An **if** statement can be followed by an optional **else if...else** statement, which is very usefull to test various conditions using single if...else if statement.

When using if , else if , else statements there are few points to keep in mind.

23 An if can have zero or one else's and it must come after any else if's.

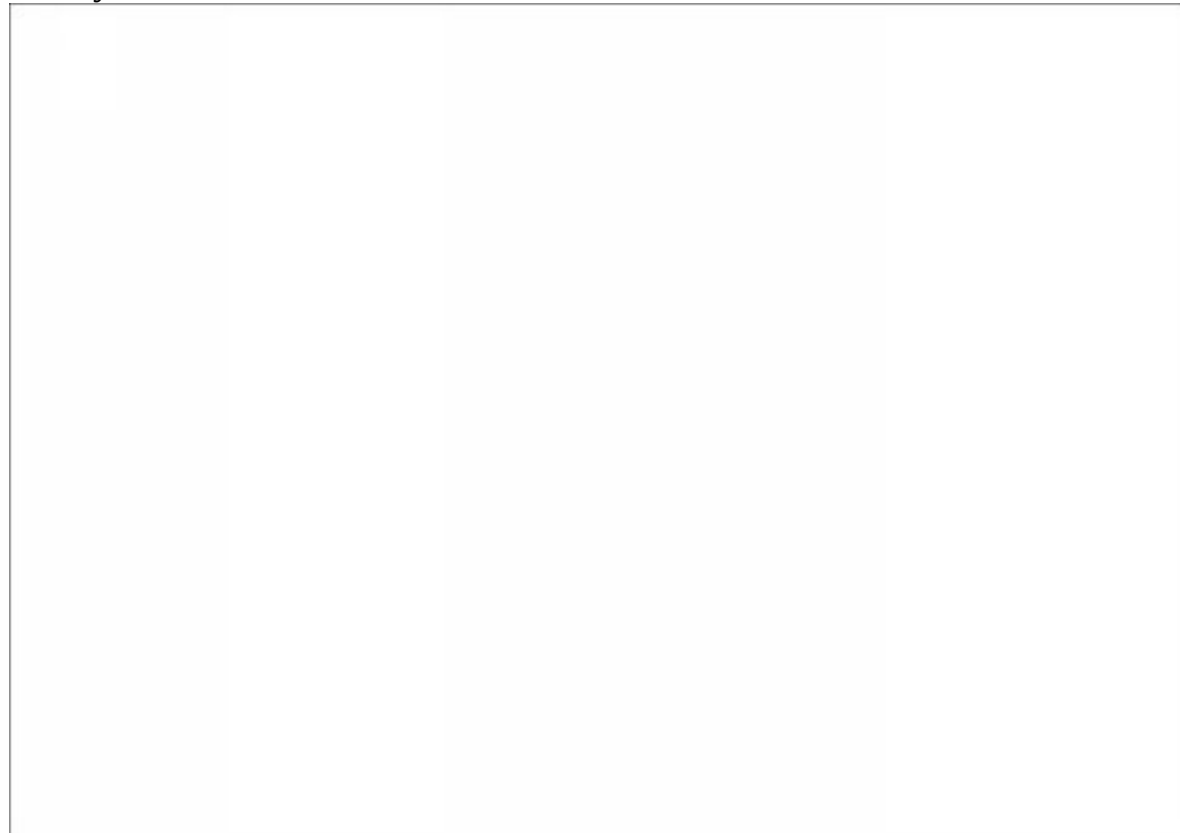
C++

An if can have zero to many else if's and they must come before the else.

Once an else if succeeds, none of the remaining else if's or else's will be tested.

Syntax

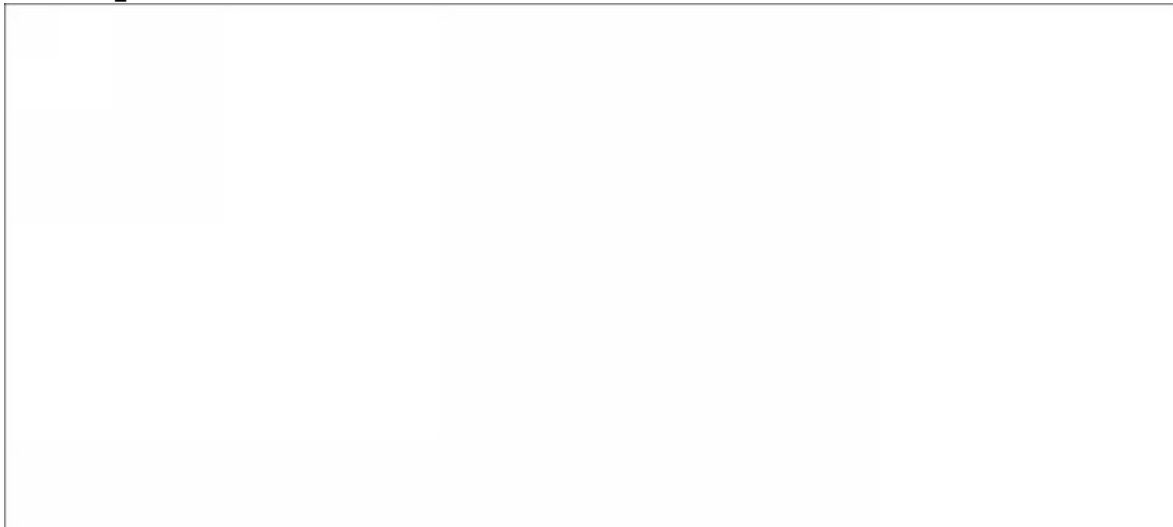
The syntax of an if...else if...else statement in C++ is:



```
if(boolean_expression 1)
{
    // Executes when the boolean expression 1 is true
}
```

```
else if( boolean_expression 2)
{
    // Executes when the boolean expression 2 is true
}
else if( boolean_expression 3)
{
    // Executes when the boolean expression 3 is true
}
else
{
    // executes when the none of the above condition is
    true.
}
```

Example



```
#include <iostream>
```

```
using namespace std;
```

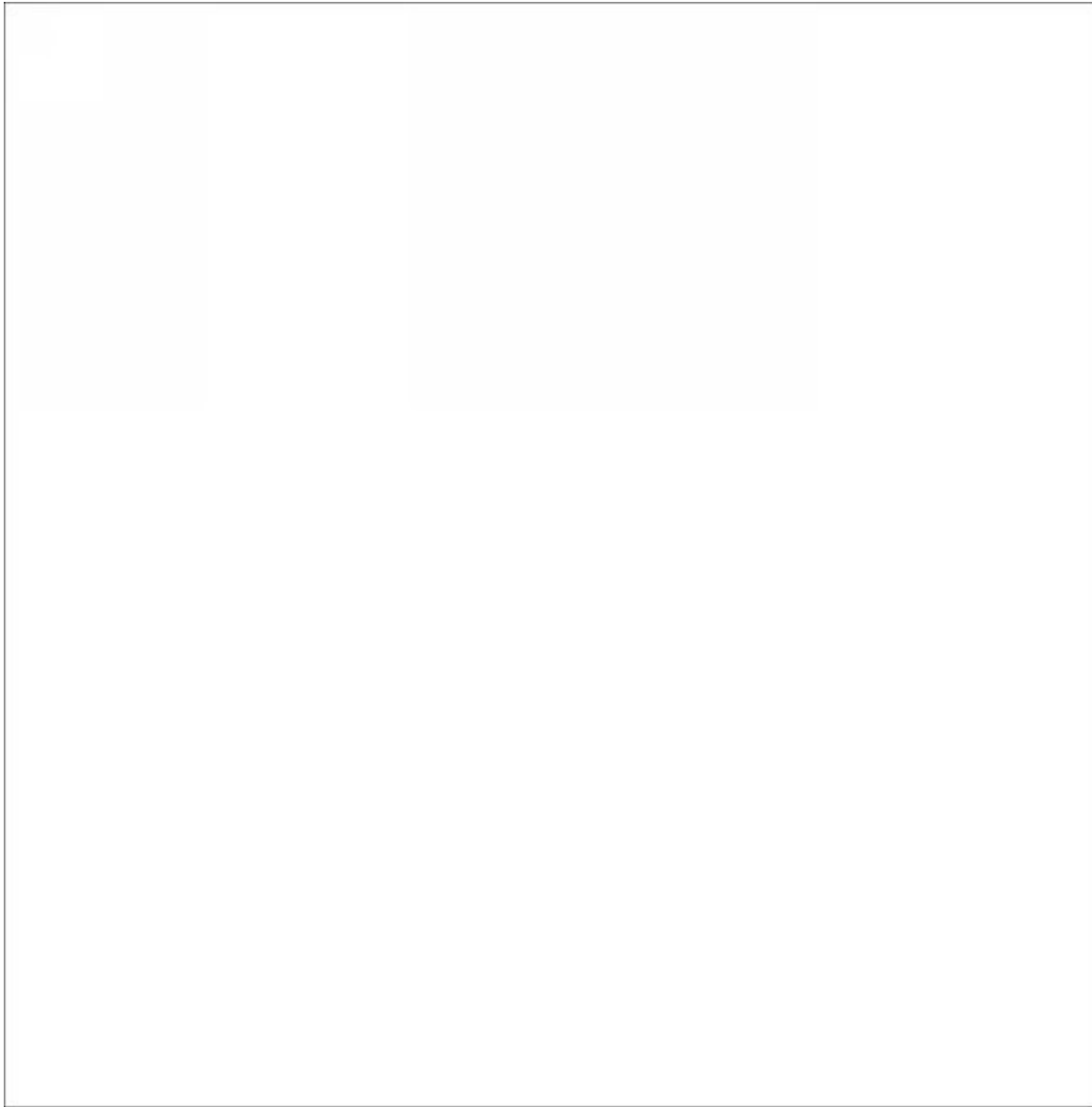
```
int main ()
```

```
{
```

```
    local variable declaration: int a = 100;
```

```
    check the boolean condition if( a == 10 )
```

C++



{

if condition is true
then print the following
cout << "Value of a is 10"
<< endl;

}


```

else if( a == 20 )
{
    if else if condition is
true cout << "Value of a
is 20" << endl;

}
else if( a == 30 )
{
    if else if condition is
true cout << "Value of a
is 30" << endl;

}
else
{
    // if none of the
conditions is true

    cout << "Value of a is not
matching" << endl;

}

cout << "Exact value of a is : " << a << endl;

return 0;

}

```

When the above code is compiled and executed, it produces the following result:

Value of a is not matching

Exact value of a is : 100

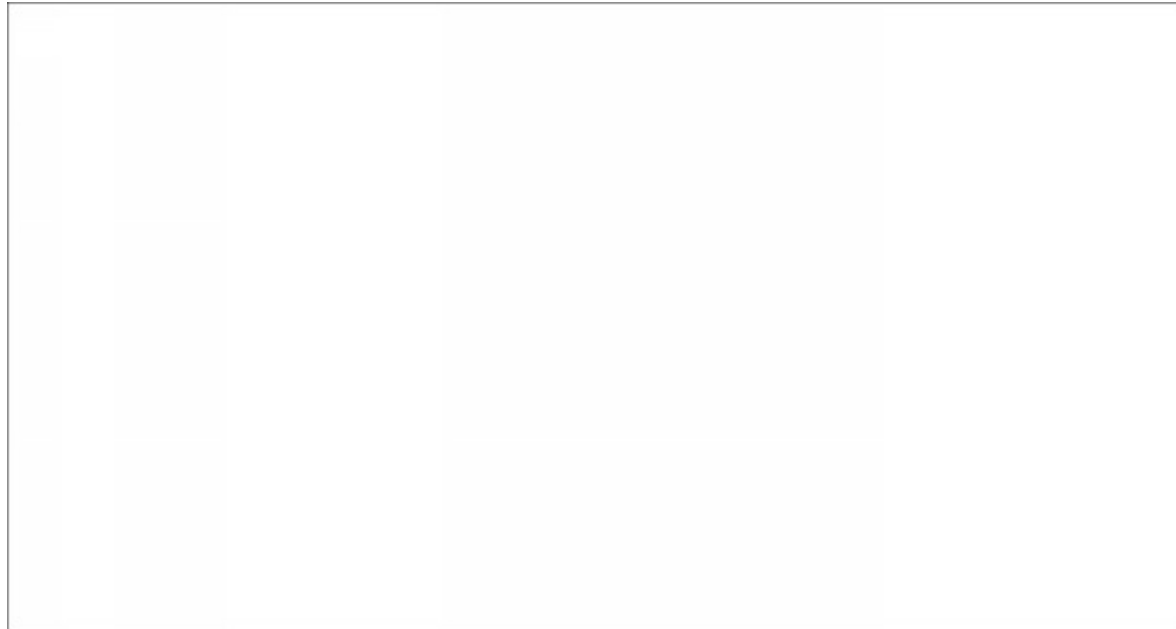
Switch Statement

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

Syntax

The syntax for a **switch** statement in C++ is as follows:

C++



```
switch(expression){
```

```
    case constant-expression    :
```

```
        statement(s);
```

```
        break; //optional
```

```
    case constant-expression    :
```

```
        statement(s);
```

```
        break; //optional
```

```
        you can have any number of case  
statements. default : //Optional
```

```
        statement(s);
```

```
}
```

The following rules apply to a switch statement:

The **expression** used in a **switch** statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.

You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.

The **constant-expression** for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.

When the variable being switched on is equal to a case, the statements following that case will execute until a **break** statement is reached.

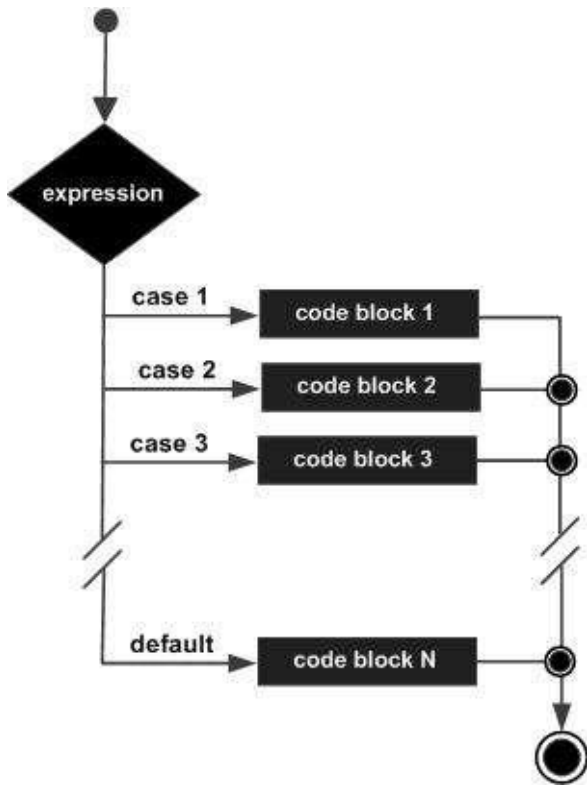
When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.

Not every case needs to contain a break. If no break appears, the flow of control will *fall through* to subsequent cases until a break is reached.

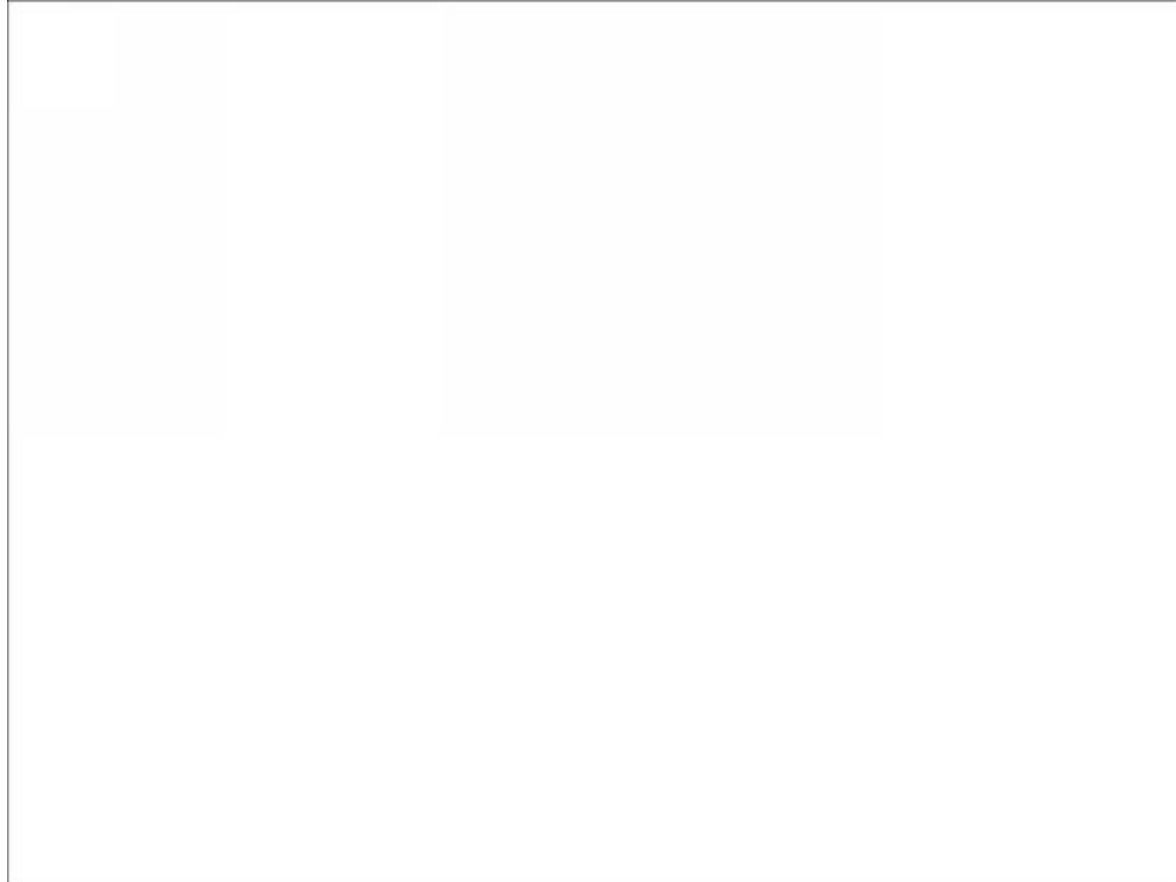
A **switch** statement can have an optional **default** case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Flow Diagram

C++



Example



```
#include <iostream>
```

```
using namespace std;
```

```
int main ()
```

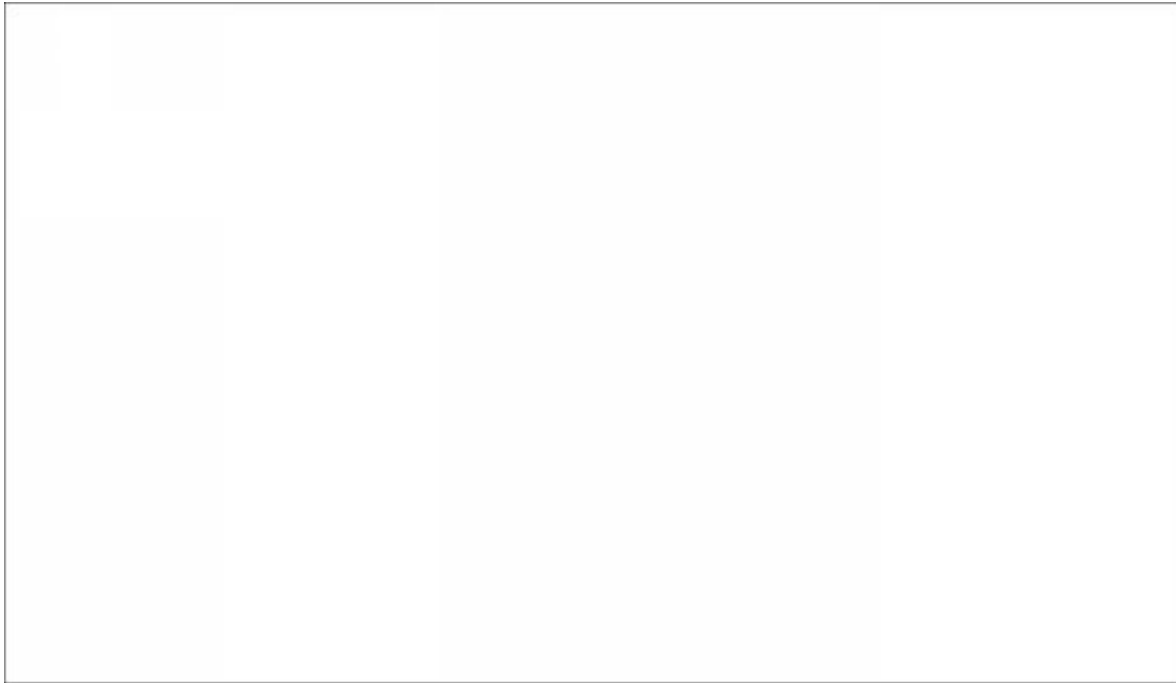
```
{
```

```
    local variable declaration: char grade = 'D';
```

```
    switch(grade)
```

```
{  
case 'A' :  
  
    cout << "Excellent!" << endl;  
    break;  
  
case 'B' :  
  
case 'C' :  
  
    cout << "Well done" << endl;  
    break;
```


C++



```
case 'D' :
```

```
    cout << "You passed" << endl;
```

```
    break;
```

```
case 'F' :
```

```
    cout << "Better try again" << endl;
```

```
    break;
```

```
default :
```

```
    cout << "Invalid grade" << endl;
```

```
}
```

```
cout << "Your grade is " << grade << endl;
```

```
return 0;
```

```
}
```

This would produce the following result:

You passed

Your grade is D

Nested if Statement

It is always legal to **nest** if-else statements, which means you can use one if or else if statement inside another if or else if statement(s).

Syntax

The syntax for a **nested if** statement is as follows:

```
if( boolean_expression 1)
```

```
{
```

```
    Executes when the boolean expression 1 is true
    if(boolean_expression 2)
```

```
    {
```

```
        Executes when the
        boolean expression 2 is
        true
```

```
}  
  
}
```

You can nest **else if...else** in the similar way as you have nested *if* statement.

C++

Example

```
#include <iostream>

using namespace std;

int main ()
```

```

{

    local variable declaration: int a = 100;

    int b = 200;

    check the boolean condition if( a == 100 )

    {

        if condition is true
        then check the following
        if( b == 200 )

        {

            if condition is true
            then print the following
            cout << "Value of a is 100
            and b is 200" << endl;

        }

    }

    cout << "Exact value of a is : " << a << endl;

    cout << "Exact value of b is : " << b << endl;

    return 0;

}

```

When the above code is compiled and executed, it produces the following result:

Value of a is 100 and b is 200

Exact value of a is : 100

Exact value of b is : 200

Nested switch Statements

It is possible to have a switch as part of the statement sequence of an outer switch. Even if the case constants of the inner and outer switch contain common values, no conflicts will arise.

C++

C++ specifies that at least 256 levels of nesting be allowed for switch statements.

Syntax

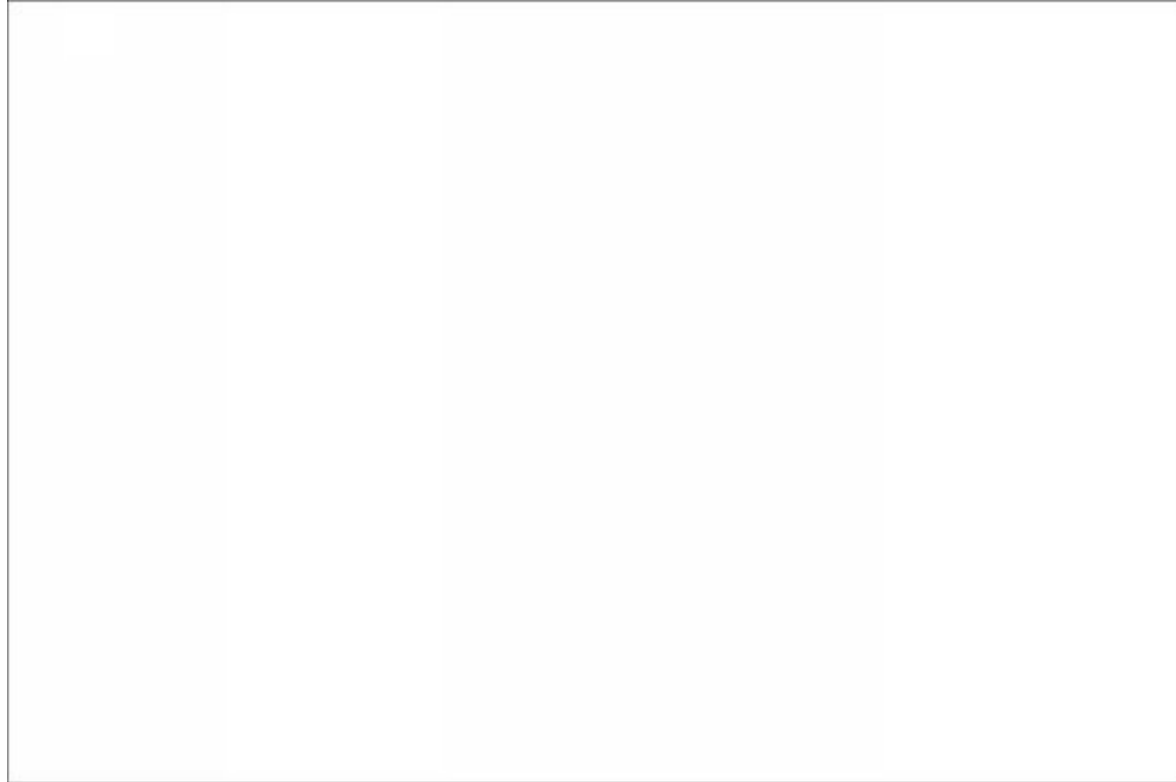
The syntax for a **nested switch** statement is as follows:



```
switch(ch1) {  
    case 'A':  
        cout << "This A is part of outer  
switch"; switch(ch2) {  
            case 'A':  
                cout << "This A is part of  
inner switch"; break;  
            case 'B': // ...  
            }  
        break;  
}
```

```
        case 'B': // ...  
    }  
}
```

Example



```
#include <iostream>  
  
using namespace std;  
  
int main ()  
{  
  
    local variable declaration: int a = 100;  
  
    int b = 200;  
  
    switch(a) {  
  
        case 100:
```



```
cout << "This is part of  
outer switch" << endl;  
switch(b) {
```

```
case 200:
```

```
cout << "This is part of  
inner switch" << endl;
```

C++



```
        }  
    }  
    cout << "Exact value of a is : " << a << endl;  
    cout << "Exact value of b is : " << b << endl;  
  
    return 0;  
}
```

This would produce the following result:



This is part of outer switch

This is part of inner switch

Exact value of a is : 100

Exact value of b is : 200

The ? : Operator

We have covered conditional operator “? :” in previous chapter which can be used to replace **if...else** statements. It has the following general form:

Exp1 ? Exp2 : Exp3;

Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon.

The value of a ‘?’ expression is determined like this: Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the value of the entire ‘?’ expression. If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the expression.



C++

A function is a group of statements that together perform a task. Every C++ program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is such that each function performs a specific task.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

The C++ standard library provides numerous built-in functions that your program can call. For example, function **strcat()** to concatenate two strings, function **memcpy()** to copy one memory location to another location, and many more functions.

A function is known with various names like a method or a sub-routine or a procedure etc.

Defining a Function

The general form of a C++ function definition is as follows:



```
return_type function_name( parameter list )  
  
{  
  
    body of the function  
  
}
```

A C++ function definition consists of a function header and a function body.

Here are all the parts of a function:

Return Type: A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.

Function Name: This is the actual name of the function. The function name and the parameter list together constitute the function signature.

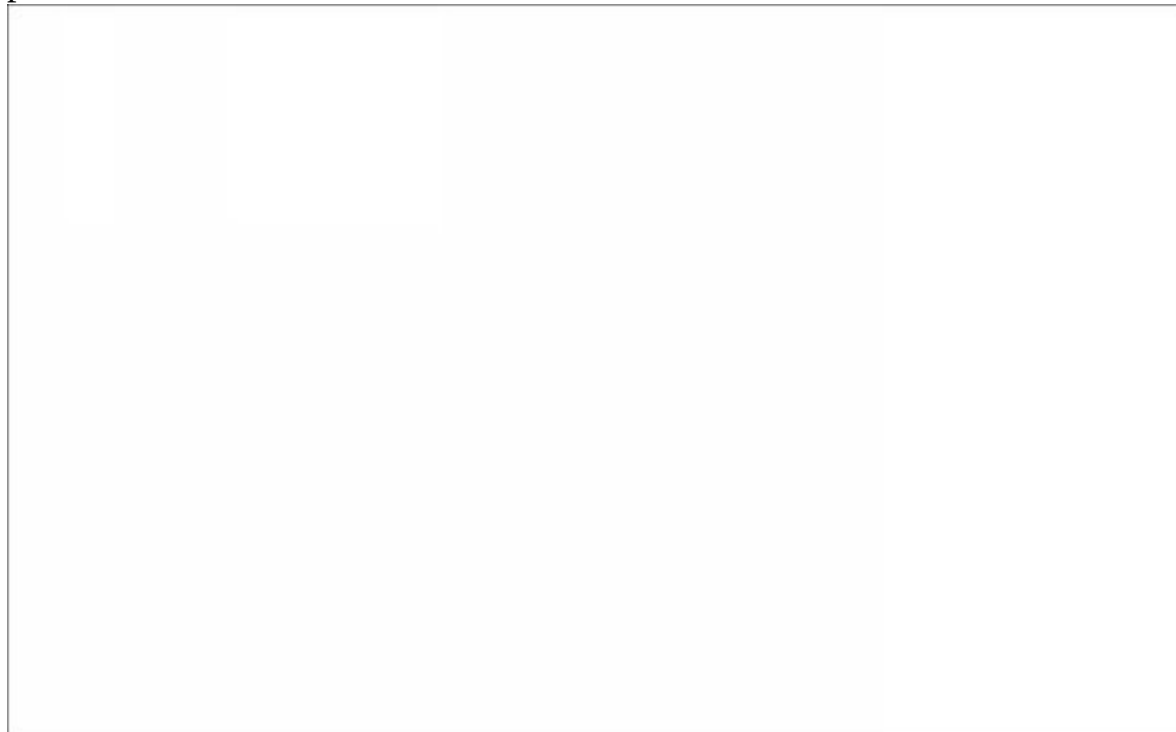
Parameters: A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

C++

Function Body: The function body contains a collection of statements that define what the function does.

Example:

Following is the source code for a function called **max()**. This function takes two parameters num1 and num2 and returns the maximum between the two:



```
// function returning the max between two numbers
```

```
int max(int num1, int num2)
```

```
{
```

```
    local variable declaration int result;
```

```
    if (num1 > num2)
```

```
        result = num1;

    else

        result = num2;

    return result;

}
```

Function Declarations

A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts:

```
return_type function_name( parameter list );
```

For the above defined function max(), following is the function declaration:

```
int max(int num1, int num2);
```

Parameter names are not important in function declaration only their type is required, so following is also valid declaration:

```
int max(int, int);
```

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

C++

Calling a Function

While creating a C++ function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function.

When a program calls a function, program control is transferred to the called function. A called function performs defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.

To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value. For example:


```
#include <iostream>
```

```
using namespace std;
```

```
// function declaration
```

```
int max(int num1, int num2);
```

```

int main ()
{
    local variable declaration: int a = 100;

    int b = 200; int ret;

    calling a function to get max value. ret = max(a,
b);

    cout << "Max value is : " << ret << endl;

    return 0;
}

```

function returning the max between two numbers int max(int num1, int num2)

```

{

```

local variable declaration

C++

```
int result;  
  
if (num1 > num2)  
    result = num1;  
else  
    result = num2;  
  
return result;  
  
}
```

I kept max() function along with main() function and compiled the source code.

While running final executable, it would produce the following result:

```
Max value is : 200
```

Function Arguments

If a function is to use arguments, it must declare variables that accept the

values of the arguments. These variables are called the **formal parameters** of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways that arguments can be passed to a function:

Call Type	Description
Call by value	This method copies the actual value of an argument into the formal parameter of the function. In this case,
	changes made to the parameter inside the function
	have no effect on the argument.
Call by pointer	This method copies the address of an argument into the formal parameter. Inside the function, the address
	is used to access the actual argument used in the call.
	This means that changes made to the parameter
	affect the argument.
Call by reference	This method copies the reference of an argument into
	the formal parameter. Inside the function, the
	reference is used to access the actual argument used
	85

C++

in the call. This means that changes made to the parameter affect the argument.

Call by Value

The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

By default, C++ uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function. Consider the function **swap()** definition as follows.

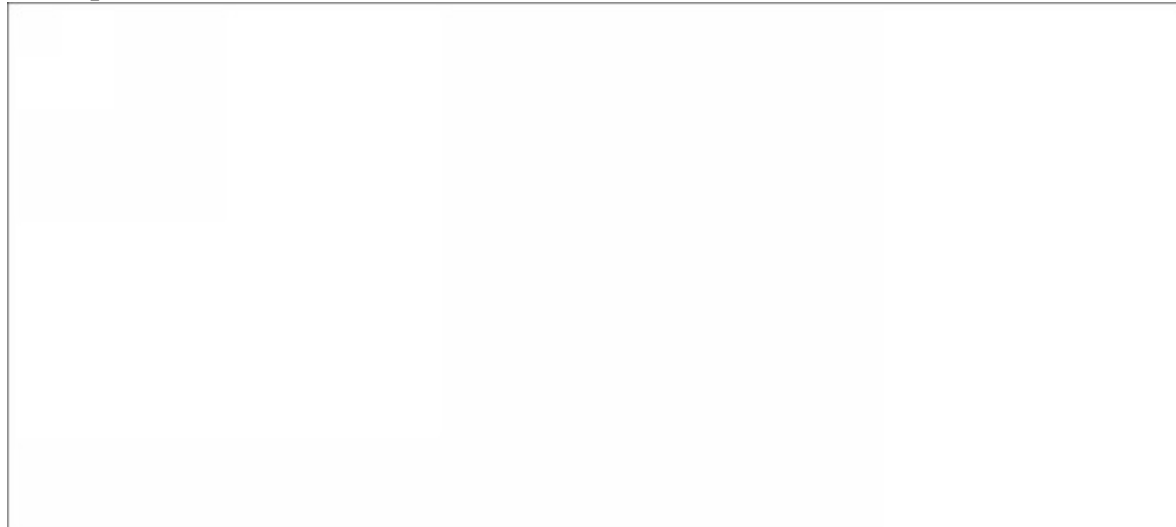


```
function definition to swap the values. void swap(int x, int y)
{
    int temp;

    temp = x; /* save the value of x */
    x = y;    /* put y into x */
    y = temp; /* put x into y */

    return;
}
```

Now, let us call the function **swap()** by passing actual values as in the following example:



```
#include <iostream>
```

```
using namespace std;
```

```
function declaration void swap(int x, int y);
```

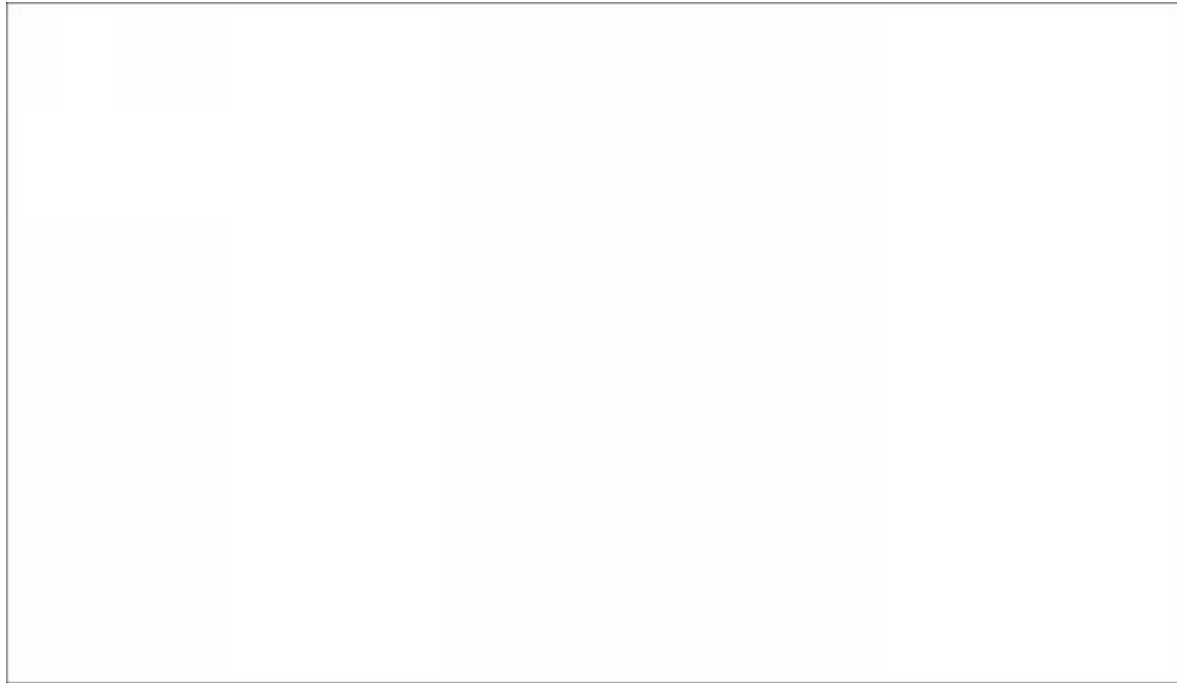
```
int main ()
```

{

local variable declaration: int a = 100;

86

C++



```
int b = 200;
```

```
cout << "Before swap, value of a :" << a << endl; cout  
<< "Before swap, value of b :" << b << endl;
```

```
calling a function to swap the values. swap(a, b);
```

```
cout << "After swap, value of a :" << a << endl; cout  
<< "After swap, value of b :" << b << endl;
```

```
return 0;
```

```
}
```

When the above code is put together in a file, compiled and executed, it produces the following result:



Before swap, value of a :100

Before swap, value of b :200

After swap, value of a :100

After swap, value of b :200

Which shows that there is no change in the values though they had been changed inside the function.

Call by Pointer

The **call by pointer** method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

To pass the value by pointer, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types as in the following function **swap()**, which exchanges the values of the two integer variables pointed to by its arguments.

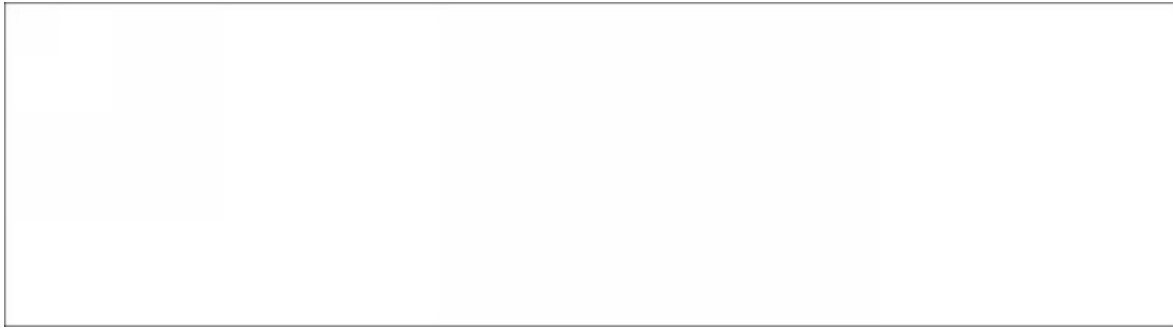


function definition to swap the values. void swap(int *x, int *y)

{

int temp;

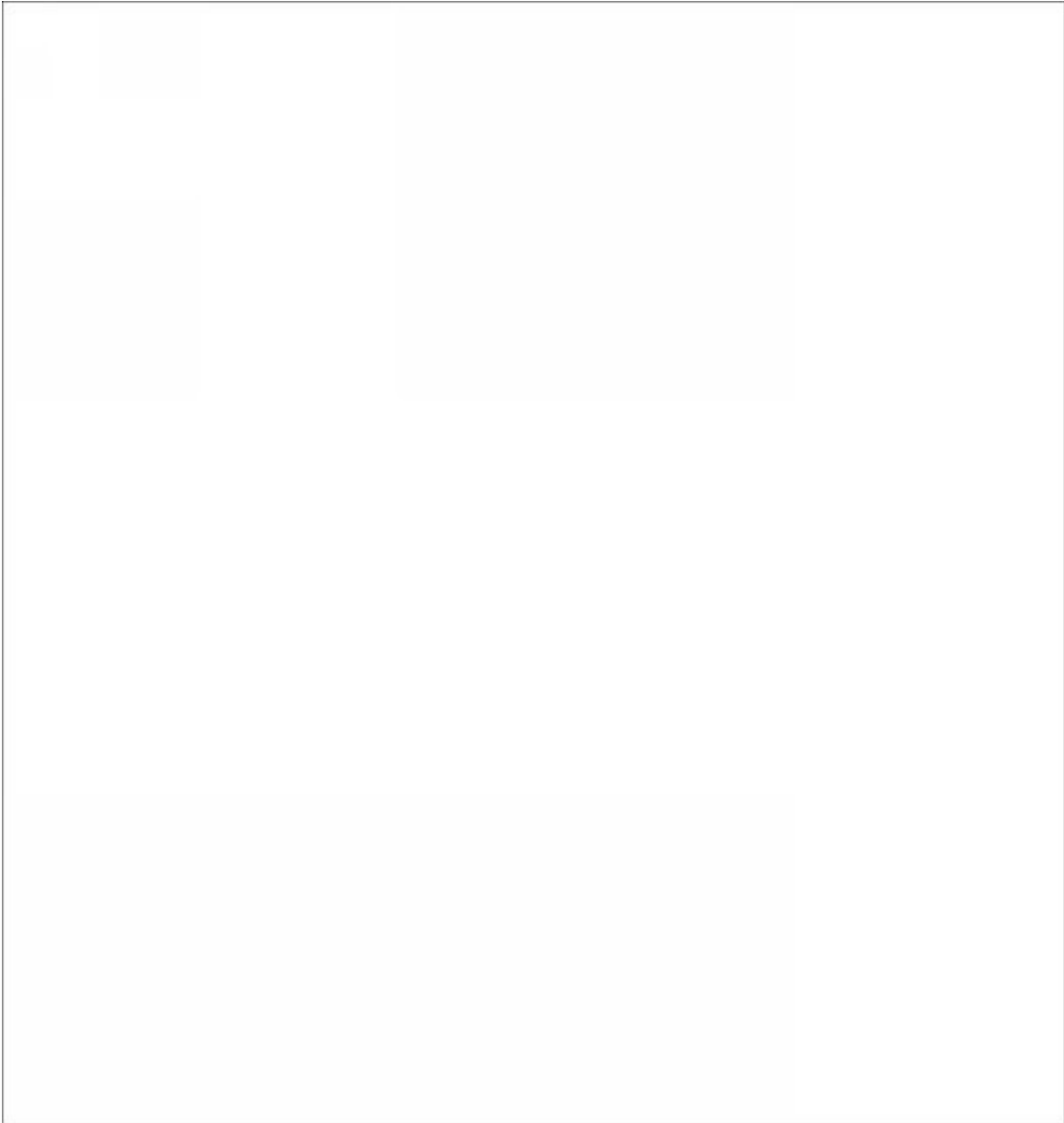
C++



```
temp = *x; /* save the value at address x */ *x = *y; /*  
put y into x */ *y = temp; /* put x into y */  
  
return;  
  
}
```

To check the more detail about C++ pointers, kindly check C++ Pointers chapter.

For now, let us call the function **swap()** by passing values by pointer as in the following example:



```
#include <iostream>
```

```
using namespace std;
```

```
function declaration
```

```
void swap(int *x, int *y);
```

```
int main ()
```

```
{
```

```
    local variable declaration: int a = 100;
```

```
    int b = 200;
```

```
    cout << "Before swap, value of a :" << a << endl; cout  
    << "Before swap, value of b :" << b << endl;
```

```
    /* calling a function to swap the values.
```

```
        &a indicates pointer to a ie. address of  
        variable a and
```

```
        &b indicates pointer to b ie. address of  
        variable b.
```

```
    */
```

```
    swap(&a, &b);
```

```
    cout << "After swap, value of a :" << a << endl; cout  
    << "After swap, value of b :" << b << endl;
```

C++

```
        return 0;

    }
```

When the above code is put together in a file, compiled and executed, it produces the following result:

Before swap, value of a :100

Before swap, value of b :200

After swap, value of a :200

After swap, value of b :100

Call by Reference

The **call by reference** method of passing arguments to a function copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

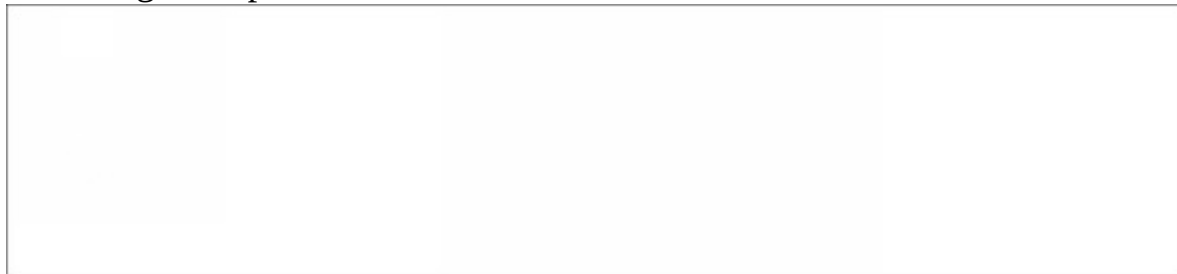
To pass the value by reference, argument reference is passed to the functions just like any other value. So accordingly you need to declare the function parameters as reference types as in the following function **swap()**, which exchanges the values of the two integer variables pointed to by its arguments.



```
function definition to swap the values. void swap(int &x, int &y)
{
    int temp;
    temp = x; /* save the value at address x */
    x = y;    /* put y into x */
    y = temp; /* put x into y */

    return;
}
```

For now, let us call the function **swap()** by passing values by reference as in the following example:



```
#include <iostream>
```

```
using namespace std;
```

```
function declaration
```

```
void swap(int &x, int &y);
```


C++

```
int main ()
```

```
{
```

```
    local variable declaration: int a = 100;
```

```
    int b = 200;
```

```
    cout << "Before swap, value of a : " << a << endl; cout  
    << "Before swap, value of b : " << b << endl;
```

```
    /* calling a function to swap the values using variable
```

```
reference.*/ swap(a, b);

cout << "After swap, value of a :" << a << endl; cout
<< "After swap, value of b :" << b << endl;

return 0;

}
```

When the above code is put together in a file, compiled and executed, it produces the following result:



Before swap, value of a :100

Before swap, value of b :200

After swap, value of a :200

After swap, value of b :100

By default, C++ uses **call by value** to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function and above mentioned example while calling max() function used the same method.

Default Values for Parameters

When you define a function, you can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function.

C++

This is done by using the assignment operator and assigning values for the arguments in the function definition. If a value for that parameter is not passed when the function is called, the default given value is used, but if a value is specified, this default value is ignored and the passed value is used instead. Consider the following example:

```
#include <iostream>
```

```
using namespace std;
```

```

int sum(int a, int b=20)
{
    int result;

    result = a + b;

    return (result);
}

int main ()
{
    local variable declaration: int a = 100;

    int b = 200; int result;

    calling a function to add the values. result =
    sum(a, b);

    cout << "Total value is :" << result << endl;

    calling a function again as follows. result =
    sum(a);

    cout << "Total value is :" << result << endl;

    return 0;
}

```

When the above code is compiled and executed, it produces the following result:

C++

--

Total value is :300

Total value is :120



C++

Normally, when we work with Numbers, we use primitive data types such as int, short, long, float and double, etc. The number data types, their possible values and number ranges have been explained while discussing C++ Data Types.

Defining Numbers in C++

You have already defined numbers in various examples given in previous chapters. Here is another consolidated example to define various types of numbers in C++:



```
#include <iostream>
```

```
using namespace std;
```

```
int main ()
```

```
{
```

```
    number definition: short s;
```

int i; long l; float f; double d;

number assignments; s = 10;

i = 1000;

l = 1000000; f = 230.47;

d = 30949.374;

number printing;

cout <<	"short	s :< << s << endl;
cout <<	"int	i :< << i << endl;
cout <<	"long	l :< <<
cout <<	"float	f :< <<

C++

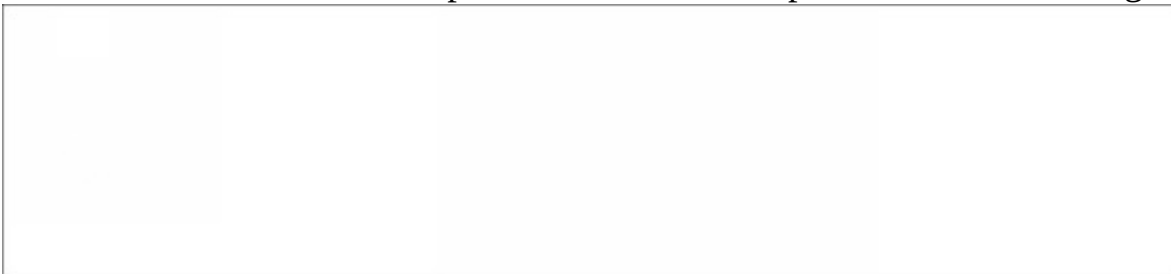


```
cout << "double d :" << d << endl;

return 0;

}
```

When the above code is compiled and executed, it produces the following result:



```
short  s :10
int    i :1000
long   l :1000000
float  f :230.47
double d :30949.4
```

Math Operations in C++

In addition to the various functions you can create, C++ also includes some useful functions you can use. These functions are available in standard C and C++ libraries and called **built-in** functions. These are functions that can be included in your program and then use.

C++ has a rich set of mathematical operations, which can be performed on

various numbers. Following table lists down some useful built-in mathematical functions available in C++.

To utilize these functions you need to include the math header file **<cmath>**.

S.N. Function & Purpose

double cos(double);

This function takes an angle (as a double) and returns the cosine.

double sin(double);

This function takes an angle (as a double) and returns the sine.

double tan(double);

This function takes an angle (as a

double) and returns the tangent.

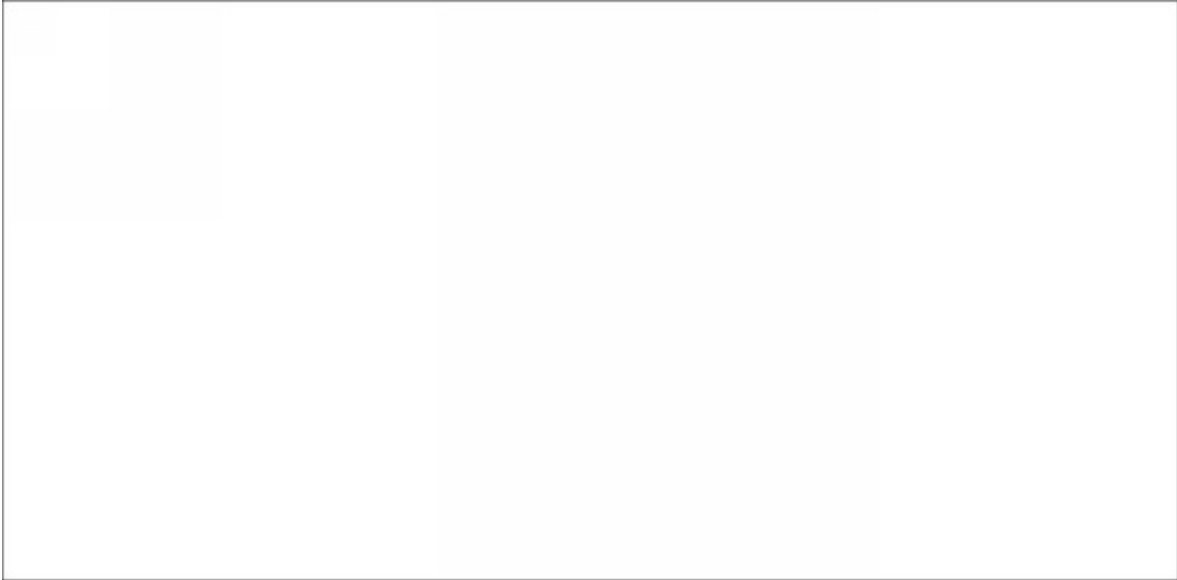
double log(double);

This function takes a number and returns the natural log of that number.

C++

5	double pow(double, double);
	The first is a number you wish to raise and the second is the power you
	wish to raise it t
6	double hypot(double, double);
	If you pass this function the length of two sides of a right triangle, it will
	return you the length of the hypotenuse.
7	double sqrt(double);
	You pass this function a number and it gives you the square root.
8	int abs(int);
	This function returns the absolute value of an integer that is passed to
	it.
9	double fabs(double);
	This function returns the absolute value of any decimal number passed
	to it.
10	double floor(double);
	Finds the integer which is less than or equal to the argument passed to
	it.

Following is a simple example to show few of the mathematical operations:



```
#include <iostream>
```

```
#include <cmath>
```

```
using namespace std;
```

```
int main ()
```

```
{
```

```
    number definition: short s = 10;
```

int	i =	-1000;
long	l =	100000;
float	f =	230.47;

C++

```
double d = 200.374;

// mathematical operations;

cout << "sin(d) :" << sin(d) << endl; cout << "abs(i) :"
<< abs(i) << endl; cout << "floor(d) :" << floor(d) <<
endl; cout << "sqrt(f) :" << sqrt(f) << endl; cout <<
"pow( d, 2) :" << pow(d, 2) << endl;

return 0;

}
```

When the above code is compiled and executed, it produces the following result:

```
sign(d) :-0.634939
```

`abs(i) :1000`

`floor(d) :200`

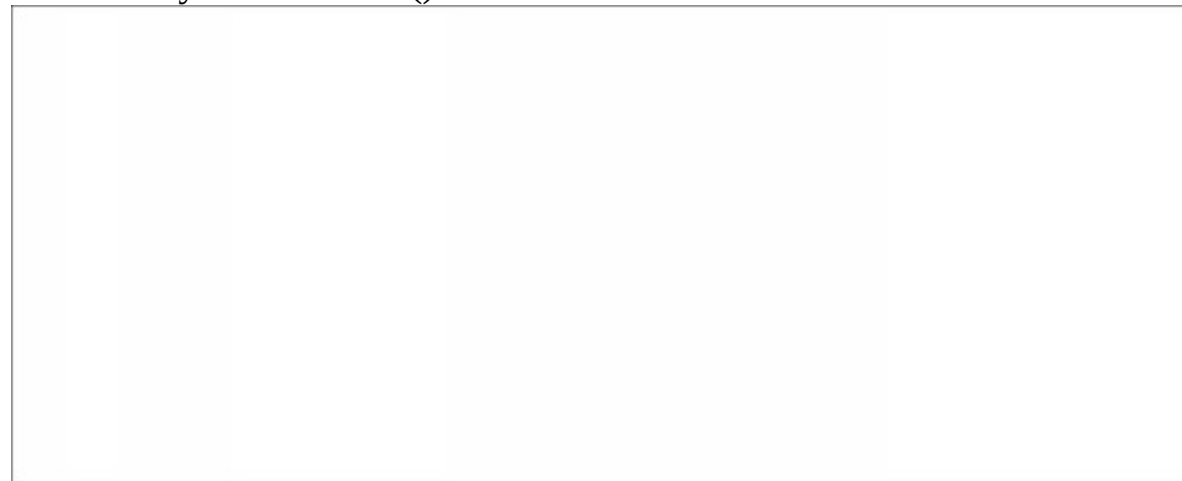
`sqrt(f) :15.1812`

`pow(d, 2) :40149.7`

Random Numbers in C++

There are many cases where you will wish to generate a random number. There are actually two functions you will need to know about random number generation. The first is **rand()**, this function will only return a pseudo random number. The way to fix this is to first call the **srand()** function.

Following is a simple example to generate few random numbers. This example makes use of **time()** function to get the number of seconds on your system time, to randomly seed the **rand()** function:



```
#include <iostream>
```

```
#include <ctime>
```

```
#include <cstdlib>
```

```
using namespace std;
```

```
int main ()
```

```
{
```

```
    int i,j;
```

C++

```
// set the seed
srand( (unsigned)time( NULL ) );

/* generate 10    random numbers. */
for( i = 0; i < 10; i++ )
{
    generate actual random number
    j= rand();

    cout <<" Random Number : " << j
    << endl;
}
```

```
        return 0;  
    }
```

When the above code is compiled and executed, it produces the following result:



Random Number : 1748144778

Random Number : 630873888

Random Number : 2134540646

Random Number : 219404170

Random Number : 902129458

Random Number : 920445370

Random Number : 1319072661

Random Number : 257938873

Random Number : 1256201101

Random Number : 580322989



C++

C++ provides a data structure, **the array**, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

Declaring Arrays

To declare an array in C++, the programmer specifies the type of the elements and the number of elements required by an array as follows:

```
type arrayName [ arraySize ];
```

This is called a single-dimension array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C++ data type. For example, to declare a 10-element array called balance of type double, use this statement:

```
double balance[10];
```

Initializing Arrays

You can initialize C++ array elements either one by one or using a single statement as follows:

```
double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets []. Following is an example to assign a single element of the array:

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write:

```
double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

You will create exactly the same array as you did in the previous example.

C++

```
balance[4] = 50.0;
```

The above statement assigns element number 5th in the array a value of 50.0. Array with 4th index will be 5th, i.e., last element because all arrays have 0 as the index of their first element which is also called base index. Following is the pictorial representation of the same array we discussed above:

	0	1	2	3	4
balance	1000.0	2.0	3.4	7.0	50.0

Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example:

```
double salary = balance[9];
```

The above statement will take 10th element from the array and assign the value to salary variable. Following is an example, which will use all the above-mentioned three concepts viz. declaration, assignment and accessing arrays:

```
#include <iostream>
```

```
using namespace std;
```

```
#include <iomanip>
```

```
using std::setw;
```

```
int main ()
```

```
{
```

```
    int n[ 10 ]; // n is an array of 10 integers
```

```
        initialize elements of array n to 0 for ( int i = 0; i <
10; i++ )
```

```
{  
  
    n[ i ] = i + 100; // set element at  
    location i to i + 100  
  
}
```

```
cout << "Element" << setw( 13 ) << "Value" << endl;
```

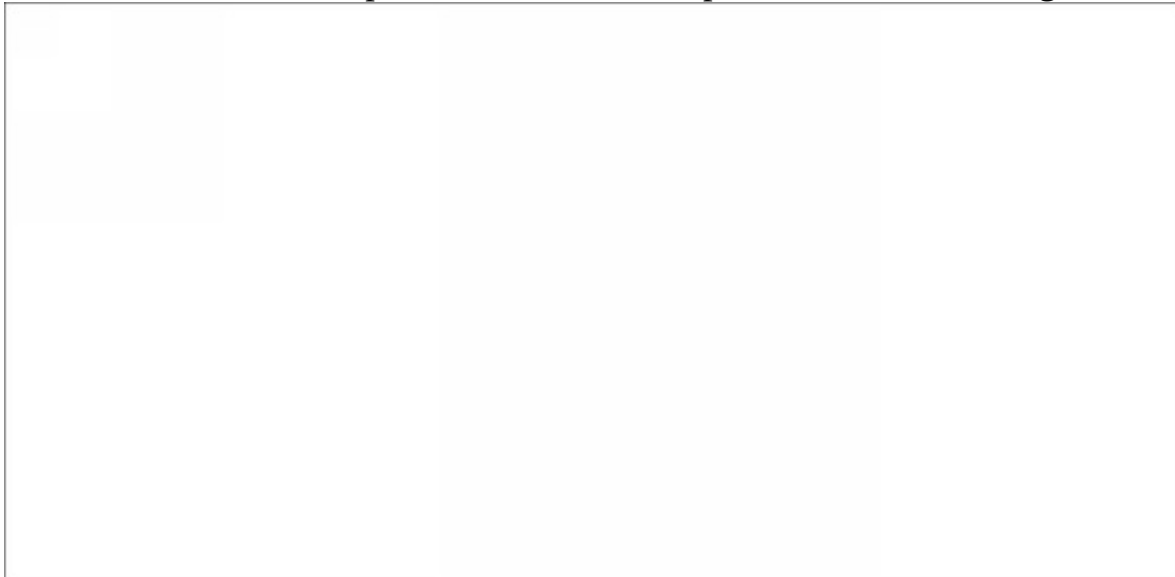
output each array element's value

C++



```
for ( int j = 0; j < 10; j++ )  
{  
    cout << setw( 7 )<< j << setw(  
13 ) << n[ j ] << endl;  
}  
  
return 0;  
  
}
```

This program makes use of **setw()** function to format the output. When the above code is compiled and executed, it produces the following result:



Element	Value
	100
	101
	102
	103
	104
	105
	106
	107
	108
	109

Arrays in C++

Arrays are important to C++ and should need lots of more detail. There are following few important concepts, which should be clear to a C++ programmer:

Concept	Description
Multi-dimensional arrays	C++ supports multidimensional arrays. The
	simplest form of the multidimensional array is
	the two-dimensional array.

Pointer to an array	You can generate a pointer to the first
	element of an array by simply specifying the
	array name, without any index.
Passing arrays to functions	You can pass to the function a pointer to an
	100

C++

	array by specifying the array's name without
	an index.
Return array from functions	C++ allows a function to return an array.

Multi-dimensional Arrays

C++ allows multidimensional arrays. Here is the general form of a multidimensional array declaration:

```
type name[size1][size2]...[sizeN];
```

For example, the following declaration creates a three dimensional 5 . 10 . 4 integer array:

```
int threedim[5][10][4];
```

Two-Dimensional Arrays

The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size x,y, you would write something as follows:

```
type arrayName [ x ][ y ];
```

Where **type** can be any valid C++ data type and **arrayName** will be a valid

C++ identifier.

A two-dimensional array can be think as a table, which will have x number of rows and y number of columns. A 2-dimensional array **a**, which contains three rows and four columns can be shown as below:

	Column 0	Column 1	Column 2	Column 3
Row 0	<code>a[0][0]</code>	<code>a[0][1]</code>	<code>a[0][2]</code>	<code>a[0][3]</code>
Row 1	<code>a[1][0]</code>	<code>a[1][1]</code>	<code>a[1][2]</code>	<code>a[1][3]</code>
Row 2	<code>a[2][0]</code>	<code>a[2][1]</code>	<code>a[2][2]</code>	<code>a[2][3]</code>

Thus, every element in array **a** is identified by an element name of the form **`a[i][j]`**, where **a** is the name of the array, and **i** and **j** are the subscripts that uniquely identify each element in **a**.

Initializing Two-Dimensional Arrays

C++

Multidimensioned arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row have 4 columns.

int a[3][4] = {				
{0, 1, 2, 3} ,	/*	initializers for row indexed by 0		*/
{4, 5, 6, 7} ,	/*	initializers for row indexed by	1	*/
{8, 9, 10, 11}	/*	initializers for row indexed by	2	*/
};				

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to previous example:

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Accessing Two-Dimensional Array Elements

An element in 2-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example:

```
int val = a[2][3];
```

The above statement will take 4th element from the 3rd row of the array. You can verify it in the above digram.

```
#include <iostream>
```

```
using namespace std;
```

```
int main ()
```

```
{
```

```
    // an array with 5 rows and 2 columns.
```

```
    int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};
```

```
        output each array element's value for ( int i = 0; i  
< 5; i++ )
```

```
            for ( int j = 0; j < 2; j++ )
```

```
            {
```

```
cout << "a[" << i << "]"["  
<< j << "]: "; cout << a[i]  
[j]<< endl;
```

```
}
```

C++

```
    return 0;  
}
```

When the above code is compiled and executed, it produces the following result:

```
a[0][0]: 0  
a[0][1]: 0  
a[1][0]: 1  
a[1][1]: 2  
a[2][0]: 2  
a[2][1]: 4  
a[3][0]: 3  
a[3][1]: 6  
a[4][0]: 4
```

`a[4][1]: 8`

As explained above, you can have arrays with any number of dimensions, although it is likely that most of the arrays you create will be of one or two dimensions.

Pointer to an Array

It is most likely that you would not understand this chapter until you go through the chapter related C++ Pointers.

So assuming you have bit understanding on pointers in C++, let us start: An array name is a constant pointer to the first element of the array. Therefore, in the declaration:

```
double balance[50];
```

balance is a pointer to `&balance[0]`, which is the address of the first element of the array `balance`. Thus, the following program fragment assigns **p** the address of the first element of **balance**:

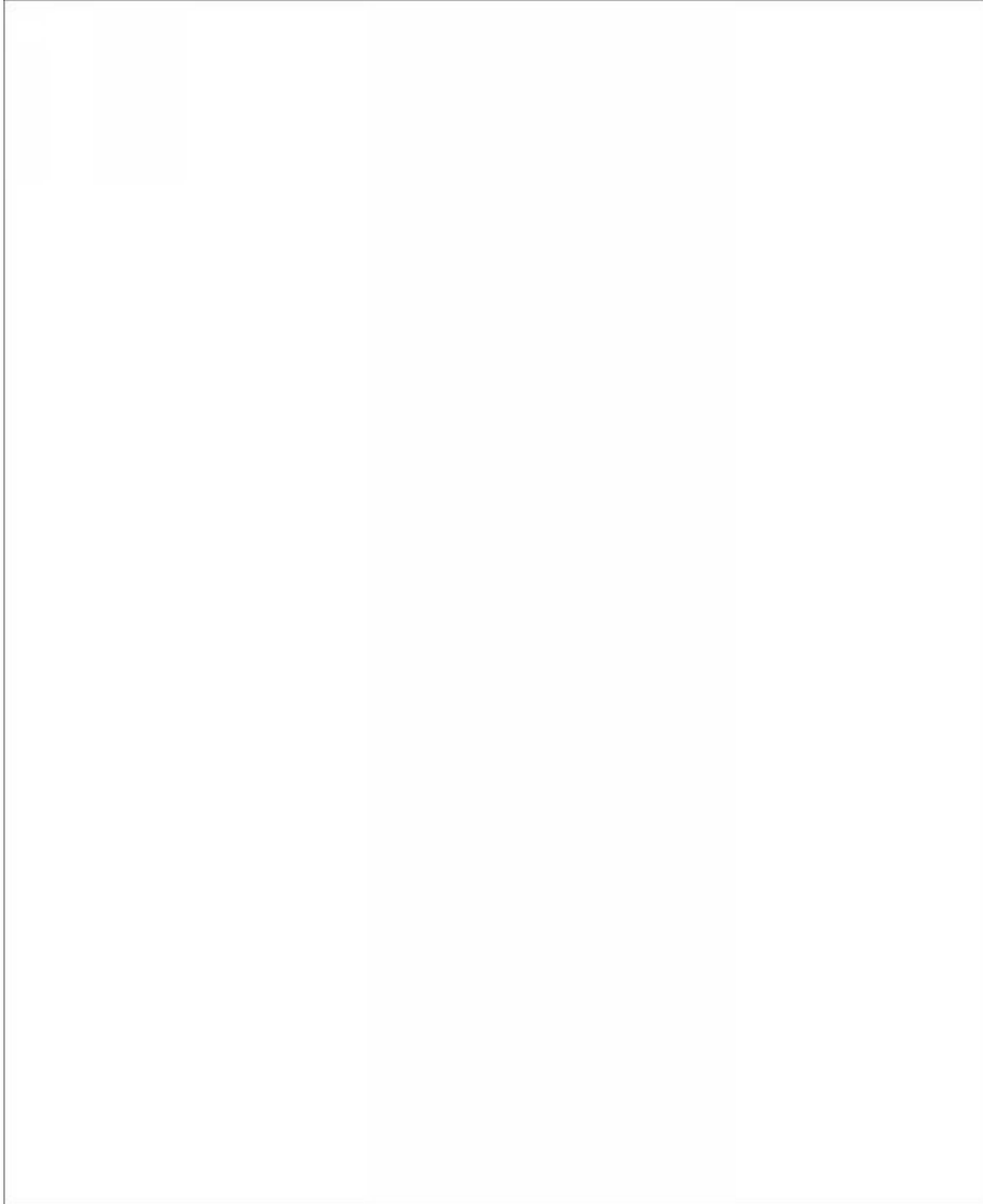
```
double *p;
```

```
double balance[10];
```

```
p = balance;
```

It is legal to use array names as constant pointers, and vice versa. Therefore, `*(balance + 4)` is a legitimate way of accessing the data at `balance[4]`.

Once you store the address of first element in `p`, you can access array elements using `*p`, `*(p+1)`, `*(p+2)` and so on. Below is the example to show all the concepts discussed above:



```

#include <iostream>

using namespace std;

int main ()
{
    // an array with 5 elements.

    double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
    double *p;

    p = balance;

    // output each array element's value

    cout << "Array values using pointer " << endl; for ( int
    i = 0; i < 5; i++ ) {

        cout << "*(p + " << i <<
        ") : ";

        cout << *(p + i) << endl;

    }

    cout << "Array values using balance as address " <<
    endl; for ( int i = 0; i < 5; i++ ) {

        cout << "*(balance + " <<
        i << ") : "; cout << *
        (balance + i) << endl;

    }

    return 0;

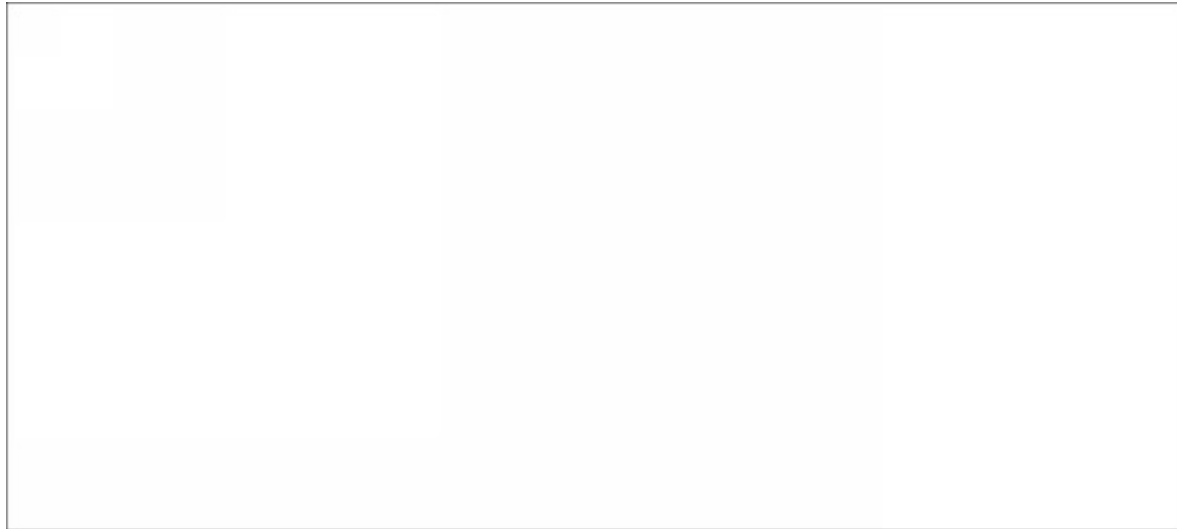
}

```

When the above code is compiled and executed, it produces the following result:

Array values using pointer

*(p + 0) : 1000



`*(p + 1) : 2`

`*(p + 2) : 3.4`

`*(p + 3) : 17`

`*(p + 4) : 50`

Array values using balance as address

`*(balance + 0) : 1000`

`*(balance + 1) : 2`

`*(balance + 2) : 3.4`

`*(balance + 3) : 17`

`*(balance + 4) : 50`

In the above example, `p` is a pointer to double which means it can store address of a variable of double type. Once we have address in `p`, then `*p` will give us value available at the address stored in `p`, as we have shown in the above example.

Passing Arrays to Functions

C++ does not allow to pass an entire array as an argument to a function. However, You can pass a pointer to an array by specifying the array's name without an index.

If you want to pass a single-dimension array as an argument in a function, you would have to declare function formal parameter in one of following three ways and all three declaration methods produce similar results because each tells the compiler that an integer pointer is going to be received.

Way-1

Formal parameters as a pointer as follows:

```
void myFunction(int *param)
{
.
.
.
}
```

Way-2

Formal parameters as a sized array as follows:

```
void myFunction(int param[10])
```

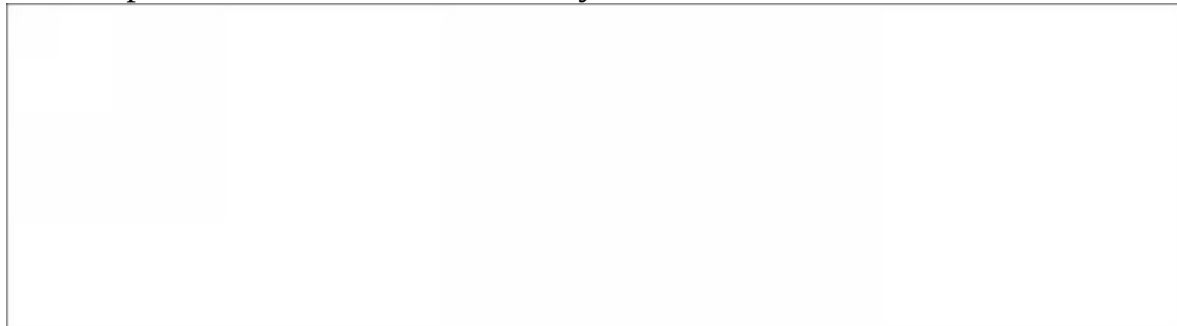
C++



```
{  
.  
.  
.  
}
```

Way-3

Formal parameters as an unsized array as follows:



```
void myFunction(int param[])  
{  
.  
.  
.  
}
```

Now, consider the following function, which will take an array as an argument along with another argument and based on the passed arguments, it will return average of the numbers passed through the array as follows:



```
double getAverage(int arr[], int size)
{
    int    i, sum = 0;
    double avg;

    for (i = 0; i < size; ++i)
    {
        sum += arr[i];
    }

    avg = double(sum) / size;
```

```
        return avg;  
    }
```

Now, let us call the above function as follows:

```
#include <iostream>
```

C++

```
using namespace std;
```

```
// function declaration:
```

```
double getAverage(int arr[], int size);
```

```
int main ()
```

```
{
```

```
    // an int array with 5 elements.
```

```
    int balance[5] = {1000, 2, 3, 17, 50}; double avg;
```

```
        pass pointer to the array as an argument. avg =  
        getAverage( balance, 5 ) ;
```

```
        output the returned value
```

```
        cout << "Average value is: " << avg << endl;
```

```
        return 0;
```

```
    }
```

When the above code is compiled together and executed, it produces the following result:

Average value is: 214.4

As you can see, the length of the array doesn't matter as far as the function is concerned because C++ performs no bounds checking for the formal parameters.

Return Array from Functions

C++ does not allow to return an entire array as an argument to a function. However, you can return a pointer to an array by specifying the array's name without an index.

If you want to return a single-dimension array from a function, you would have to declare a function returning a pointer as in the following example:

```
int * myFunction()
```

```
{
```

```
.
```


C++

.

.

}

Second point to remember is that C++ does not advocate to return the address of a local variable to outside of the function so you would have to define the local variable as **static** variable.

Now, consider the following function, which will generate 10 random numbers and return them using an array and call this function as follows:

```
#include <iostream>
```

```
#include <ctime>
```

```
using namespace std;
```

```
function to generate and retrun random numbers. int * getRandom( )
```

```

{

    static int r[10];

    set the seed

    srand( (unsigned)time( NULL ) );

    for (int i = 0; i < 10; ++i)
    {

        r[i] = rand();

        cout << r[i] << endl;

    }

    return r;

}

```

main function to call above defined function. int main ()

```

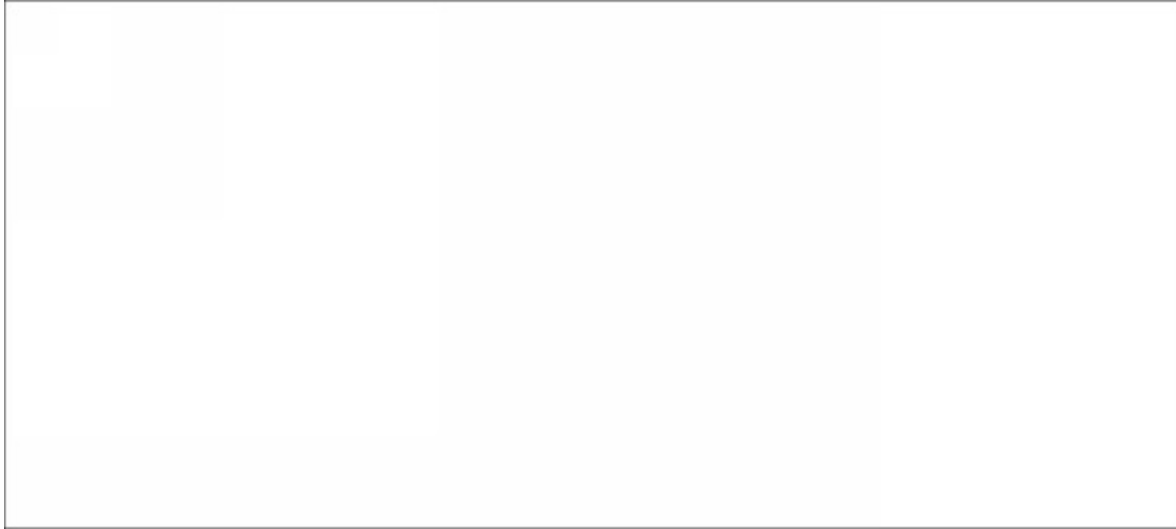
{

    a pointer to an int.

    int *p;

```

C++



```
p = getRandom();  
for ( int i = 0; i < 10; i++ )  
{  
  
    cout << "(p + " << i <<  
    "): ";  
  
    cout << *(p + i) << endl;  
  
}  
  
return 0;  
  
}
```

When the above code is compiled together and executed, it produces result something as follows:



624723190

1468735695

807113585

976495677

613357504

1377296355

1530315259

1778906708

1820354158

667126415

*(p + 0) : 624723190

*(p + 1) : 1468735695

*(p + 2) : 807113585

*(p + 3) : 976495677

*(p + 4) : 613357504

*(p + 5) : 1377296355

*(p + 6) : 1530315259

*(p + 7) : 1778906708

*(p + 8) : 1820354158

*(p + 9) : 667126415

C++ provides following two types of string representations:

The C-style character string.

The string class type introduced with Standard C++. The C-Style Character String

The C-style character string originated within the C language and continues to be supported within C++. This string is actually a one-dimensional array of characters which is terminated by a **null** character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a **null**.

The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

If you follow the rule of array initialization, then you can write the above statement as follows:

```
char greeting[] = "Hello";
```

Following is the memory presentation of above defined string in C/C++:

Index	0	1	2	3	4	5
Variable	H	e	l	l	o	\0
Address	0x23451	0x23452	0x23453	0x23454	0x23455	0x23456

Actually, you do not place the null character at the end of a string constant. The C++ compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print above-mentioned string:

```
#include <iostream>
```

```
using namespace std;
```


C++

```
int main ()
{
    char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

    cout << "Greeting message: ";
    cout << greeting << endl;

    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Greeting message: Hello
```

C++ supports a wide range of functions that manipulate null-terminated strings:

S.N.	Function & Purpose
1	strcpy(s1, s2); Copies string s2 into string s1.
2	strcat(s1, s2); Concatenates string s2 onto the end of string s1.
3	strlen(s1); Returns the length of string s1.

4	strcmp(s1, s2);
	Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than
	0 if s1>s2.
5	strchr(s1, ch);
	Returns a pointer to the first occurrence of character ch in string s1.

strstr(s1, s2);

C++

Returns a pointer to the first occurrence of string s2 in string s1.

Following example makes use of few of the above-mentioned functions:

```
#include <iostream>
```

```
#include <cstring>
```

```
using namespace std;
```

```
int main ()
```

```

{
    char str1[10] = "Hello";
    char str2[10] = "World";
    char str3[10];
    int len ;

    // copy str1 into str3
    strcpy( str3, str1);

    cout << "strcpy( str3, str1) : " << str3 << endl;

    // concatenates str1 and str2
    strcat( str1, str2);
    cout << "strcat( str1, str2): " << str1 << endl;

    // total length of str1 after concatenation
    len = strlen(str1);
    cout << "strlen(str1) : " << len << endl;

    return 0;
}

```

When the above code is compiled and executed, it produces result something as follows:

```


```

```

strcpy( str3, str1) : Hello

```

C++

```
strcat( str1, str2): HelloWorld
```

```
strlen(str1) : 10
```

The String Class in C++

The standard C++ library provides a **string** class type that supports all the operations mentioned above, additionally much more functionality. Let us check the following example:

```
#include <iostream>
```

```
#include <string>
```

```
using namespace std;
```

```
int main ()
```



```

{
    string str1 = "Hello";
    string str2 = "World";
    string str3;
    int len ;

    // copy str1 into str3
    str3 = str1;

    cout << "str3 : " << str3 << endl;

    // concatenates str1 and str2
    str3 = str1 + str2;

    cout << "str1 + str2 : " << str3 << endl;

    // total length of str3 after concatenation
    len = str3.size();

    cout << "str3.size() : " << len << endl;

    return 0;
}

```

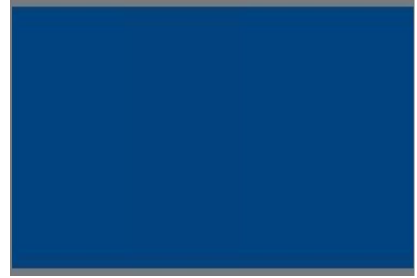
When the above code is compiled and executed, it produces result something as follows:

C++

str3 : Hello

str1 + str2 : HelloWorld

str3.size() : 10



c

The C++ Standard Library can be categorized into two parts:

The Standard Function Library: This library consists of general-purpose, stand-alone functions that are not part of any class. The function library is inherited from C.

The Object Oriented Class Library: This is a collection of classes and associated functions.

Standard C++ Library incorporates all the Standard C libraries also, with small additions and changes to support type safety.

The Standard Function Library

The standard function library is divided into the following categories:

I/O,

String and character handling,

Mathematical,
Time, date, and localization,
Dynamic allocation,
Miscellaneous,
Wide-character functions

The Object Oriented Class Library

Standard C++ Object Oriented Library defines an extensive set of classes that provide support for a number of common activities, including I/O, strings, and numeric processing. This library includes the following:

The Standard C++ I/O Classes

The String Class

The Numeric Classes

The STL Container Classes

The STL Algorithms

The STL Function Objects

The STL Iterators

The STL Allocators

C++

Structure of a program

Probably the best way to start learning a programming language is by writing a program. Therefore, here is our first program:



```
// my first program in C++    Hello World!
```

```
#include <iostream>
```

```
using namespace std;
```

```
int main ()  
{
```

```
    cout << "Hello World!";  
    return 0;
```

```
}
```

The first panel shows the source code for our first program. The second one shows the result of the program once compiled and executed. The way to edit and compile a program depends on the compiler you are using. Depending on whether it has a Development Interface or not and on its version. Consult the compilers section and the manual or help included with your compiler if you have doubts on how to compile a C++ console program.

The previous program is the typical program that programmer apprentices write for the first time, and its result is the printing on screen of the "Hello World!" sentence. It is one of the simplest programs that can be written in C++, but it already contains the fundamental components that every C++ program has. We are going to look line by line at the code we have just written:

// my first program in C++

This is a comment line. All lines beginning with two slash signs (//) are considered comments and do not have any

effect on the behavior of the program. The programmer can use them to include short explanations or observations within the source code itself. In this case, the line is a brief description of what our program is.

#include <iostream>

Lines beginning with a hash sign (#) are directives for the preprocessor. They are not regular code lines with expressions but indications for the compiler's preprocessor. In this case the directive `#include <iostream>` tells the preprocessor to include the `iostream` standard file. This specific file (`iostream`) includes the declarations of the basic standard input-output library in C++, and it is included because its functionality is going to be used later in the program.

using namespace std;

All the elements of the standard C++ library are declared within what is called a namespace, the namespace with the name *std*. So in order to access its functionality we declare with this expression that we will be using these entities. This line is very frequent in C++ programs that use the standard library, and in fact it will be included in most of the source codes included in these tutorials.

int main ()

This line corresponds to the beginning of the definition of the main function. The main function is the point by where all C++ programs start their execution, independently of its location within the source code. It does not matter whether there are other functions with other names defined before or after it - the instructions contained within this function's definition will always be the first ones to be executed in any C++ program. For that same reason, it is essential that all C++ programs have a main function.

The word `main` is followed in the code by a pair of parentheses (`()`). That is because it is a function declaration: In C++, what differentiates a function declaration from other types of expressions are these parentheses that follow its name. Optionally, these parentheses may enclose a list of parameters within them.

Right after these parentheses we can find the body of the main function enclosed in braces (`{}`). What is contained within these braces is what the function does when it is executed.



```
cout << "Hello World!";
```

This line is a C++ statement. A statement is a simple or compound expression that can actually produce some effect. In fact, this statement performs the only action that generates a visible effect in our first program.

cout represents the standard output stream in C++, and the meaning of the entire statement is to insert a sequence of characters (in this case the Hello World sequence of characters) into the standard output stream (which usually is the screen).

cout is declared in the iostream standard file within the std namespace, so that's why we needed to include that specific file and to declare that we were going to use this specific namespace earlier in our code.

Notice that the statement ends with a semicolon character (;). This character is used to mark the end of the statement and in fact it must be included at the end of all expression statements in all C++ programs (one of the most common syntax errors is indeed to forget to include some semicolon after a statement).

```
return 0;
```

The return statement causes the main function to finish. return may be followed by a return code (in our example is followed by the return code 0). A return code of 0 for the main function is generally interpreted as the program worked as expected without any errors during its execution. This is the most usual way to end a C++ console program.

You may have noticed that not all the lines of this program perform actions when the code is executed. There were lines containing only comments (those beginning by //). There were lines with directives for the compiler's preprocessor (those beginning by #). Then there were lines that began the declaration of a function (in this case, the main function) and, finally lines with statements (like the insertion into cout), which were all included within the block delimited by the braces ({}) of the main function.

The program has been structured in different lines in order to be more readable, but in C++, we do not have strict rules on how to separate instructions in different lines. For example, instead of



```
int main ()  
{  
  
    cout << " Hello World!";  
    return 0;  
  
}
```

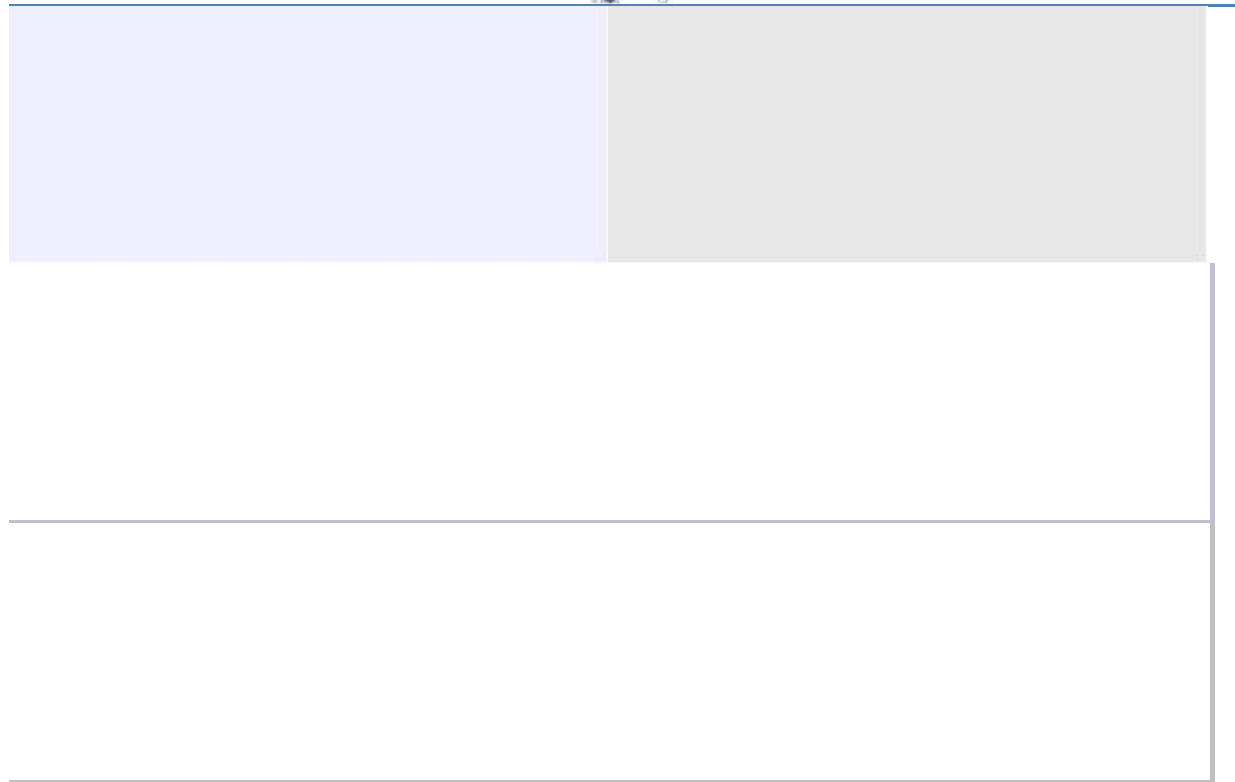
We could have written:

```
int main () { cout << "Hello World!"; return 0; }
```

All in just one line and this would have had exactly the same meaning as the previous code.

In C++, the separation between statements is specified with an ending semicolon (;) at the end of each one, so the separation in different code lines does not matter at all for this purpose. We can write many statements per line or write a single statement that takes many code lines. The division of code in different lines serves only to make it more legible and schematic for the humans that may read it.

Let us add an additional instruction to our first program:



<code>// my second program in C++</code>	Hello World! I'm a C++ program
--	--------------------------------

```
#include <iostream>
using namespace std;

int main ()
{
    cout << "Hello World! ";
    cout << "I'm a C++ program";

    return 0;
}
```

In this case, we performed two insertions into cout in two different statements. Once again, the separation in different lines of code has been done just to give greater readability to the program, since main could have been perfectly valid defined this way:

```
int main () { cout << " Hello World! "; cout << " I'm a C++ program "; return 0; }
```

We were also free to divide the code into more lines if we considered it more convenient:

```
int main ()
{
    cout <<
        "Hello World!";
    cout
        "I'm a C++ program"; return 0;
}
```

And the result would again have been exactly the same as in the previous examples.

Preprocessor directives (those that begin by #) are out of this general rule since they are not statements. They are lines read and processed by the preprocessor and do not produce any code by themselves. Preprocessor directives must be specified in their own line and do not have to end with a semicolon (;).

Comments

Comments are parts of the source code disregarded by the compiler. They simply do nothing. Their purpose is only to allow the programmer to insert notes or descriptions embedded within the source code.

C++ supports two ways to insert comments:

```
// line comment
```

```
/* block comment */
```

The first of them, known as line comment, discards everything from where the pair of slash signs (//) is found up to the end of that same line. The second one, known as block comment, discards everything

between the `/*` characters and the first appearance of the `*/` characters, with the possibility of including more than one line.

We are going to add comments to our second program:



/* my second program in C++		Hello World! I'm a C++ program
with more comments */		
#include <iostream>		
using namespace std;		
int main ()		
{		
cout << "Hello World! ";	// prints Hello	
World!		
cout << "I'm a C++ program";	// prints I'm a	
C++ program		
return 0;		
}		

If you include comments within the source code of your programs without using the comment characters combinations //, /* or */, the compiler will take them as if they were C++ expressions, most likely causing one or several error messages when you compile it.



Variables. Data Types.

The usefulness of the "Hello World" programs shown in the previous section is quite questionable. We had to write several lines of code, compile them, and then execute the resulting program just to obtain a simple sentence written on the screen as result. It certainly would have been much faster to type the output sentence by ourselves. However, programming is not limited only to printing simple texts on the screen. In order to go a little further on and to become able to write programs that perform useful tasks that really save us work we need to introduce the concept of variable.

Let us think that I ask you to retain the number 5 in your mental memory, and then I ask you to memorize also the number 2 at the same time. You have just stored two different values in your memory. Now, if I ask you to add 1 to the first number I said, you should be retaining the numbers 6 (that is 5+1) and 2 in your memory. Values that we could now for example subtract and obtain 4 as result.

The whole process that you have just done with your mental memory is a simile of what a computer can do with two variables. The same process can be expressed in C++ with the following instruction set:



```
a = 5;  
b = 2;  
  
a = a + 1;  
result = a - b;
```

Obviously, this is a very simple example since we have only used two small integer values, but consider that your computer can store millions of numbers like these at the same time and conduct sophisticated mathematical operations with them.

Therefore, we can define a variable as a portion of memory to store a determined value.

Each variable needs an identifier that distinguishes it from the others, for example, in the previous code the variable identifiers were a, b and result, but we could have called the variables any names we wanted to invent, as long as they were valid identifiers.

Identifiers

A valid identifier is a sequence of one or more letters, digits or underscore characters (_). Neither spaces nor punctuation marks or symbols can be part of an identifier. Only letters, digits and single underscore

characters are valid. In addition, variable identifiers always have to begin with a letter. They can also begin with an underline character (_), but in some cases these may be reserved for compiler specific keywords or external identifiers, as well as identifiers containing two successive underscore characters anywhere. In no case they can begin with a digit.

Another rule that you have to consider when inventing your own identifiers is that they cannot match any keyword of the C++ language nor your compiler's specific ones, which are *reserved keywords*. The standard reserved keywords are:

asm, auto, bool, break, case, catch, char, class, const, const_cast, continue, default, delete, do, double, dynamic_cast, else, enum, explicit, export, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, operator, private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_cast, struct, switch, template, this, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t, while

Additionally, alternative representations for some operators cannot be used as identifiers since they are reserved words under some circumstances:

and, and_eq, bitand, bitor, compl, not, not_eq, or, or_eq, xor, xor_eq



Your compiler may also include some additional specific reserved keywords.

Very important: The C++ language is a "case sensitive" language. That means that an identifier written in capital letters is not equivalent to another one with the same name but written in small letters. Thus, for example, the RESULT variable is not the same as the result variable or the Result variable. These are three different variable identifiers.

Fundamental data types

When programming, we store the variables in our computer's memory, but the computer has to know what kind of data we want to store in them, since it is not going to occupy the same amount of memory to store a simple number than to store a single letter or a large number, and they are not going to be interpreted the same way.

The memory in our computers is organized in bytes. A byte is the minimum amount of memory that we can manage in C++. A byte can store a relatively small amount of data: one single character or a small integer (generally an integer between 0 and 255). In addition, the computer can manipulate more complex data types that come from grouping several bytes, such as long numbers or non-integer numbers.

Next you have a summary of the basic fundamental data types in C++, as well as the range of values that can be represented with each one:

.....

Name	Description	Size*	Range*
char	Character or small integer.	1byte	signed: -128 to 127
			unsigned: 0 to 255
short int	Short Integer.	2bytes	signed: -32768 to 32767
(short)			unsigned: 0 to 65535
			signed: -2147483648 to
int	Integer.	4bytes	2147483647
			unsigned: 0 to 4294967295
			signed: -2147483648 to
long int (long)	Long integer.	4bytes	2147483647
			unsigned: 0 to 4294967295
bool	Boolean value. It can take one of two values: true	1byte	true or false
	or false.		

float	Floating point number.	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	Double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	Long double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
wchar_t	Wide character.	2 or 4	1 wide character
		bytes	

The values of the columns **Size** and **Range** depend on the system the program is compiled for. The values shown above are those found on most 32-bit systems. But for other systems, the general specification is that int has the natural size suggested by the system architecture (one "word") and the four integer types char, short, int and long must each one be at least as large as the one preceding it, with char being always 1 byte in size. The same applies to the floating point types float, double and long double, where each one must provide at least as much precision as the preceding one.

Declaration of variables

In order to use a variable in C++, we must first declare it specifying which data type we want it to be. The syntax to declare a new variable is to write the specifier of the desired data type (like int, bool, float...) followed by a valid variable identifier. For example:



```
int a;
```

```
float mynumber;
```

These are two valid declarations of variables. The first one declares a variable of type `int` with the identifier `a`. The second one declares a variable of type `float` with the identifier `mynumber`. Once declared, the variables `a` and `mynumber` can be used within the rest of their scope in the program.

If you are going to declare more than one variable of the same type, you can declare all of them in a single statement by separating their identifiers with commas. For example:

```
int a, b, c;
```

This declares three variables (`a`, `b` and `c`), all of them of type `int`, and has exactly the same meaning as:

```
int a;
```

```
int b;
```

```
int c;
```

The integer data types `char`, `short`, `long` and `int` can be either signed or unsigned depending on the range of numbers needed to be represented. Signed types can represent both positive and negative values, whereas unsigned types can only represent positive values (and zero). This can be specified by using either the specifier `signed` or the specifier `unsigned` before the type name. For example:

```
unsigned short int NumberOfSisters;
```

```
signed int MyAccountBalance;
```

By default, if we do not specify either signed or unsigned most compiler settings will assume the type to be signed, therefore instead of the second declaration above we could have written:

```
int MyAccountBalance;
```

with exactly the same meaning (with or without the keyword `signed`)

An exception to this general rule is the char type, which exists by itself and is considered a different fundamental data type from signed char and unsigned char, thought to store characters. You should use either signed or unsigned if you intend to store numerical values in a char-sized variable.


short and long can be used alone as type specifiers. In this case, they refer to their respective integer fundamental types: short is equivalent to short int and long is equivalent to long int. The following two variable declarations are equivalent:



```
short Year;
```

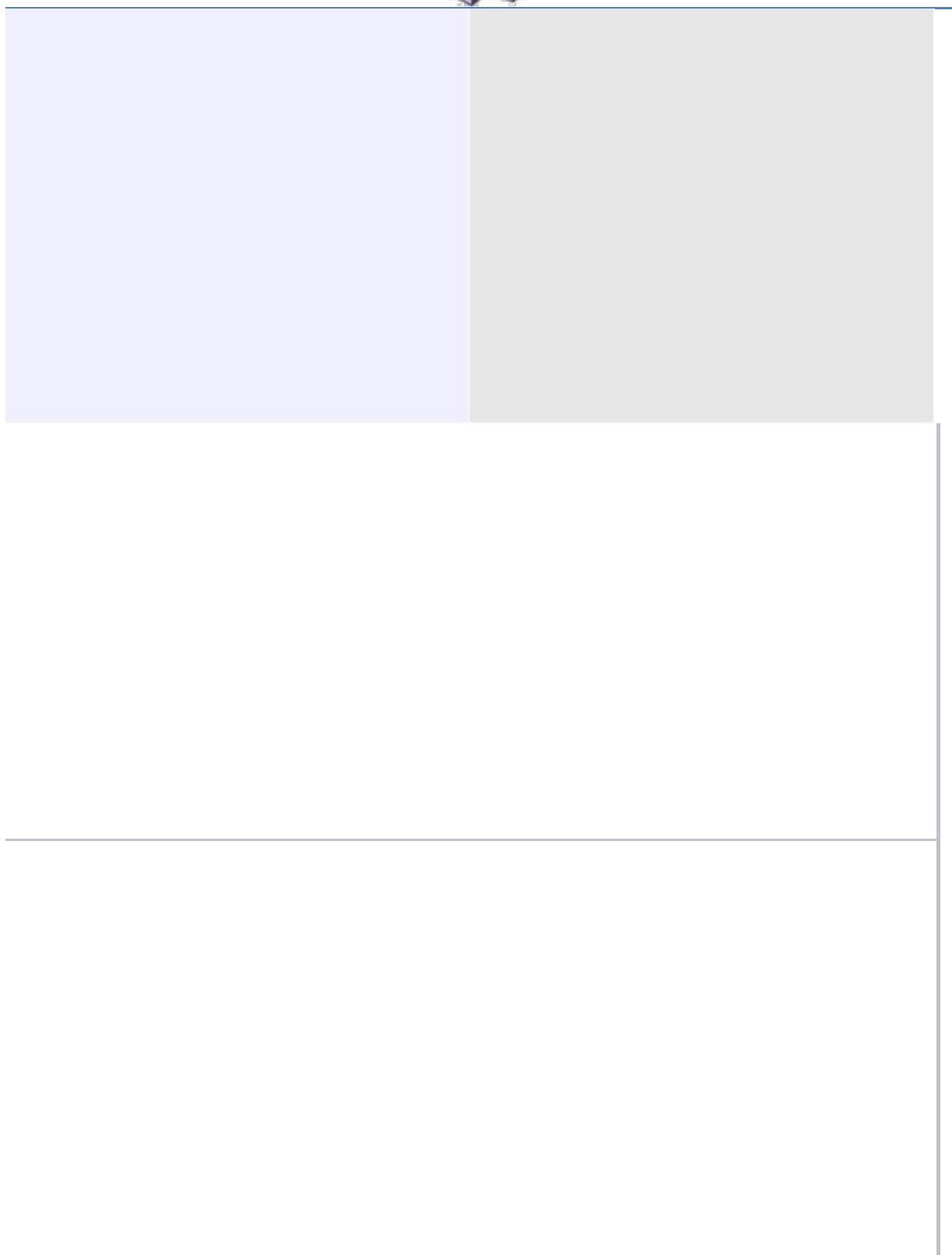
```
short int Year;
```

Finally, signed and unsigned may also be used as standalone type specifiers, meaning the same as signed int and unsigned int respectively. The following two declarations are equivalent:



```
unsigned NextYear;  
unsigned int NextYear;
```

To see what variable declarations look like in action within a program, we are going to see the C++ code of the example about your mental memory proposed at the beginning of this section:



```
// operating with variables 4
```

```
#include <iostream>
using namespace std;

int main ()
{
    // declaring variables: int a, b;

    int result;

    // process:
    a = 5;

    b = 2;
    a = a + 1;

    result = a - b;

    // print out the result: cout << result;

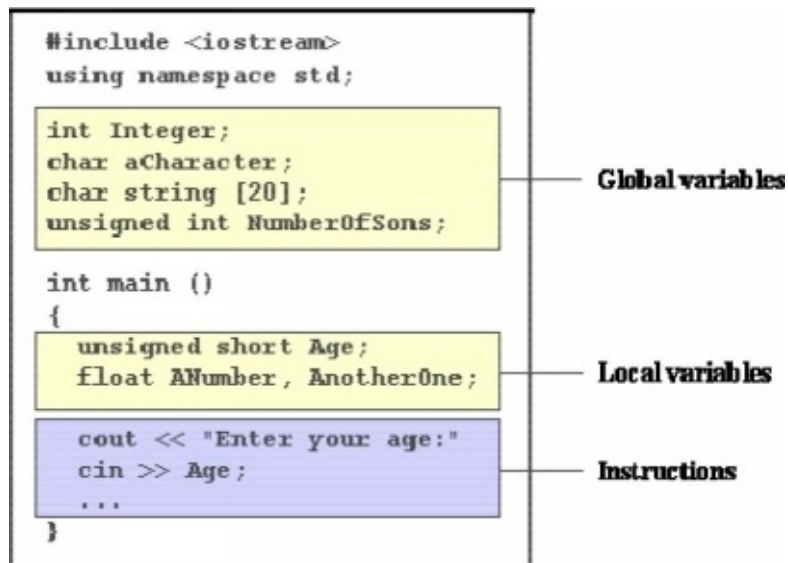
    // terminate the program: return 0;
}
```

Do not worry if something else than the variable declarations themselves looks a bit strange to you. You will see the rest in detail in coming sections.

Scope of variables

All the variables that we intend to use in a program must have been declared with its type specifier in an earlier point in the code, like we did in the previous code at the beginning of the body of the function main when we declared that a, b, and result were of type int.

A variable can be either of global or local scope. A global variable is a variable declared in the main body of the source code, outside all functions, while a local variable is one declared within the body of a function or a block.



Global variables can be referred from anywhere in the code, even inside functions, whenever it is after its declaration.



The scope of local variables is limited to the block enclosed in braces ({}) where they are declared. For example, if they are declared at the beginning of the body of a function (like in function main) their scope is between its declaration point and the end of that function. In the example above, this means that if another function existed in addition to main, the local variables declared in main could not be accessed from the other function and vice versa.

Initialization of variables

When declaring a regular local variable, its value is by default undetermined. But you may want a variable to store a concrete value at the same moment that it is declared. In order to do that, you can initialize the variable. There are two ways to do this in C++:

The first one, known as c-like, is done by appending an equal sign followed by the value to which the variable will be initialized:

```
type identifier = initial_value ;
```

For example, if we want to declare an int variable called a initialized with a value of 0 at the moment in which it is declared, we could write:

```
int a = 0;
```

The other way to initialize variables, known as constructor initialization, is done by enclosing the initial value between parentheses ():

```
type identifier (initial_value) ;
```

For example:

```
int a (0);
```

Both ways of initializing variables are valid and equivalent in C++.

// initialization of variables	6
#include <iostream>	
using namespace std;	
int main ()	
{	
int a=5;	// initial value = 5

int b(2);	// initial value = 2	
int result;	// initial value	
undetermined		
a = a + 3;		
result = a - b;		
cout << result;		
return 0;		
}		

Introduction to strings

Variables that can store non-numerical values that are longer than one single character are known as strings.

The C++ language library provides support for strings through the standard string class. This is not a fundamental type, but it behaves in a similar way as fundamental types do in its most basic usage.



A first difference with fundamental data types is that in order to declare and use objects (variables) of this type we need to include an additional header file in our source code: `<string>` and have access to the `std` namespace (which we already had in all our previous programs thanks to the `using namespace std;` statement).

// my first string	This is a string
#include <iostream>	
#include <string>	
using namespace std;	
int main ()	
{	
string mystring = "This is a string";	
cout << mystring;	
return 0;	
}	

As you may see in the previous example, strings can be initialized with any valid string literal just like numerical type variables can be initialized to any valid numerical literal. Both initialization formats are valid with strings:


```
string mystring = "This is a string";
```

```
string mystring ("This is a string");
```

Strings can also perform all the other basic operations that fundamental data types can, like being declared without an initial value and being assigned values during execution:

--	--

// my first string	This is the initial string content
#include <iostream>	This is a different string content
#include <string>	
using namespace std;	

```
int main ()
{
    string mystring;

    mystring = "This is the initial string content";
    cout << mystring << endl;

    mystring = "This is a different string content";
    cout << mystring << endl;

    return 0;
}
```

For more details on C++ strings, you can have a look at the [string class reference](#).



Constants

Constants are expressions with a fixed value.

Literals

Literals are used to express particular values within the source code of a program. We have already used these previously to give concrete values to variables or to express messages we wanted our programs to print out, for example, when we wrote:

```
a = 5;
```

the 5 in this piece of code was a literal constant.

Literal constants can be divided in Integer Numerals, Floating-Point Numerals, Characters, Strings and Boolean Values.

Integer Numerals

```
1776
```

```
707
```

```
-273
```

They are numerical constants that identify integer decimal values. Notice that to express a numerical constant we do not have to write quotes (") nor any special character. There is no doubt that it is a constant: whenever we write 1776 in a program, we will be referring to the value 1776.

In addition to decimal numbers (those that all of us are used to use every day) C++ allows the use as literal constants of octal numbers (base 8) and hexadecimal numbers (base 16). If we want to express an octal number we have to precede it with a 0 (zero character). And in order to express a hexadecimal number we have to precede it with the characters 0x (zero, x). For example, the following literal constants are all equivalent to each other:

```
// decimal
```

0113	//	octal
0x4b	//	hexadecimal

All of these represent the same number: 75 (seventy-five) expressed as a base-10 numeral, octal numeral and hexadecimal numeral, respectively.

Literal constants, like variables, are considered to have a specific data type. By default, integer literals are of type `int`. However, we can force them to either be unsigned by appending the `u` character to it, or long by appending `l`:



`// int`

`75u // unsigned int`

`75l // long`

`75ul // unsigned long`

In both cases, the suffix can be specified using either upper or lowercase letters.

Floating Point Numbers

They express numbers with decimals and/or exponents. They can include either a decimal point, an `e` character (that expresses "by ten at the Xth height", where `X` is an integer value that follows the `e` character), or both a decimal point and an `e` character:



3.14159	// 3.14159
6.02e23	// 6.02 x 10^23
1.6e-19	// 1.6 x 10^-19
3.0	// 3.0

These are four valid numbers with decimals expressed in C++. The first number is PI, the second one is the number of Avogadro, the third is the electric charge of an electron (an extremely small number) -all of them approximated- and the last one is the number three expressed as a floating-point numeric literal.

The default type for floating point literals is double. If you explicitly want to express a float or long double numerical literal, you can use the f or l suffixes respectively:

```
3.14159L    // long double
```

```
6.02e23f    // float
```

Any of the letters that can be part of a floating-point numerical constant (e, f, l) can be written using either lower or uppercase letters without any difference in their meanings.

Character and string literals

There also exist non-numerical constants, like:

```
'z'
```

```
'p'
```

```
"Hello world"
```

```
"How do you do?"
```

The first two expressions represent single character constants, and the following two represent string literals composed of several characters. Notice that to represent a single character we enclose it between single quotes (') and to express a string (which generally consists of more than one character) we enclose it

between double quotes (").

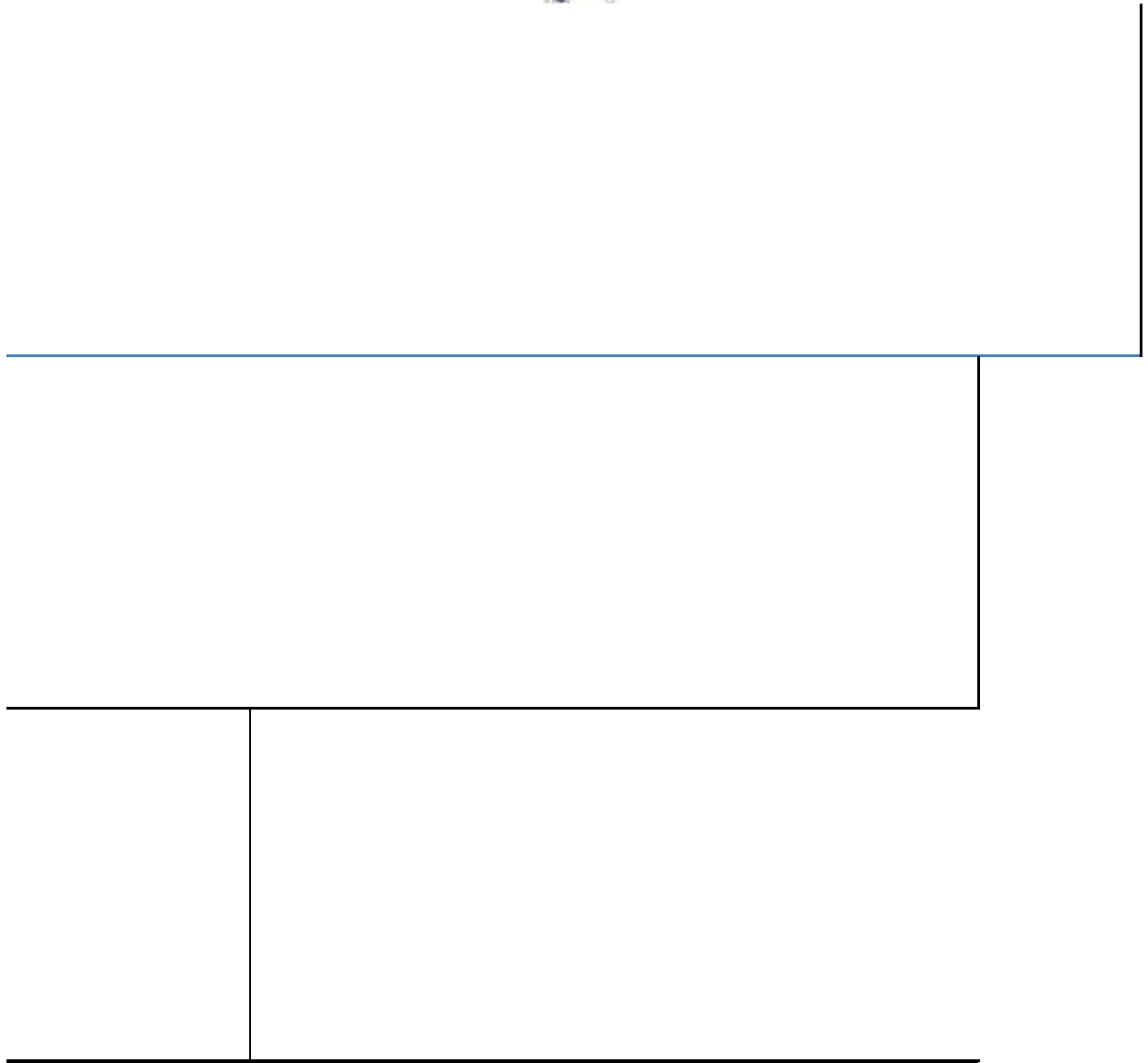
When writing both single character and string literals, it is necessary to put the quotation marks surrounding them to distinguish them from possible variable identifiers or reserved keywords. Notice the difference between these two expressions:



The diagram illustrates the memory layout for two expressions. The first expression, `x`, is shown as a single memory cell. The second expression, `'x'`, is shown as two memory cells: the first cell contains the character `x` and the second cell contains the null terminator `\0`. This visualizes how a character constant is stored as a string in memory.

`x` alone would refer to a variable whose identifier is `x`, whereas `'x'` (enclosed within single quotation marks) would refer to the character constant `'x'`.

Character and string literals have certain peculiarities, like the escape codes. These are special characters that are difficult or impossible to express otherwise in the source code of a program, like newline (`\n`) or tab (`\t`). All of them are preceded by a backslash (`\`). Here you have a list of some of such escape codes:



`\n` newline

`\r` carriage return

`\t` tab

`\v` vertical tab

`\b` backspace

`\f` form feed (page feed)

`\a` alert (beep)

`\'` single quote (')

`\"` double quote (")

`\?` question mark (?)

`\\` backslash (\) For example:

```
"\n"
"\t"
"Left \t Right"
"one\ntwo\nthree"
```

Additionally, you can express any character by its numerical ASCII code by writing a backslash character (`\`) followed by the ASCII code expressed as an octal (base-8) or hexadecimal (base-16) number. In the first case (octal) the digits must immediately follow the backslash (for example `\23` or `\40`), in the second case (hexadecimal), an `x` character must be written before the digits themselves (for example `\x20` or `\x4A`).

String literals can extend to more than a single line of code by putting a backslash sign (`\`) at the end of each unfinished line.

```
"string expressed in \
two lines"
```

You can also concatenate several string constants separating them by one or several blank spaces, tabulators, newline or any other valid blank character:

```
"this forms" "a single" "string" "of characters"
```

Finally, if we want the string literal to be explicitly made of wide characters (`wchar_t`), instead of narrow characters (`char`), we can precede the constant with the `L` prefix:

```
L"This is a wide character string"
```

Wide characters are used mainly to represent non-English or exotic character sets.

Boolean literals

There are only two valid Boolean values: true and false. These can be expressed in C++ as values of type bool by using the Boolean literals true and false.

Defined constants (#define)

You can define your own names for constants that you use very often without having to resort to memory-consuming variables, simply by using the #define preprocessor directive. Its format is:



#define identifier value

For example:

```
#define PI 3.14159
```

```
#define NEWLINE '\n'
```

This defines two new constants: PI and NEWLINE. Once they are defined, you can use them in the rest of the code as if they were any other regular constant, for example:

// defined constants: calculate circumference	31.4159
#include <iostream>	
using namespace std;	
#define PI 3.14159	
#define NEWLINE '\n'	
int main ()	
{	
double r=5.0;	// radius
double circle;	
circle = 2 * PI * r;	
cout << circle;	
cout << NEWLINE;	
return 0;	
}	

In fact the only thing that the compiler preprocessor does when it encounters #define directives is to literally replace any occurrence of their identifier (in the previous example, these were PI and NEWLINE) by the code to which they have been defined (3.14159 and '\n' respectively).

The #define directive is not a C++ statement but a directive for the preprocessor; therefore it assumes the entire line as the directive and does not require a semicolon (;) at its end. If you append a semicolon character (;) at the end, it will also be appended in all occurrences within the body of the program that the preprocessor replaces.

Declared constants (const)

With the const prefix you can declare constants with a specific type in the same way as you would do with a variable:

```
const int pathwidth = 100;  
const char tabulator = '\t';
```

Here, pathwidth and tabulator are two typed constants. They are treated just like regular variables except that their values cannot be modified after their definition.



Operators

Once we know of the existence of variables and constants, we can begin to operate with them. For that purpose, C++ integrates operators. Unlike other languages whose operators are mainly keywords, operators in C++ are mostly made of signs that are not part of the alphabet but are available in all keyboards. This makes C++ code shorter and more international, since it relies less on English words, but requires a little of learning effort in the beginning.

You do not have to memorize all the content of this page. Most details are only provided to serve as a later reference in case you need it.

Assignment (=)

The assignment operator assigns a value to a variable.

```
a = 5;
```

This statement assigns the integer value 5 to the variable a. The part at the left of the assignment operator (=) is known as the *lvalue* (left value) and the right one as the *rvalue* (right value). The lvalue has to be a variable whereas the rvalue can be either a constant, a variable, the result of an operation or any combination of these. The most important rule when assigning is the *right-to-left* rule: The assignment operation always takes place from right to left, and never the other way:

```
a = b;
```

This statement assigns to variable a (the lvalue) the value contained in variable b (the rvalue). The value that was stored until this moment in a is not considered at all in this operation, and in fact that value is lost.

Consider also that we are only assigning the value of b to a at the moment of the assignment operation. Therefore a later change of b will not affect the new value of a.

For example, let us have a look at the following code - I have included the evolution of the content stored in the variables as comments:

// assignment operator			a:4 b:7
#include <iostream>			
using namespace std;			
int main ()			

{			
int a, b;	// a:?, b:?		
a = 10;	// a:10, b:?		
b = 4;	// a:10, b:4		
a = b;	// a:4, b:4		
b = 7;	// a:4, b:7		
cout << "a:";			
cout << a;			
cout << " b:";			
cout << b;			
return 0;			
}			

This code will give us as result that the value contained in a is 4 and the one contained in b is 7. Notice how a was not affected by the final modification of b, even though we declared a = b earlier (that is because of the *right-to-left rule*).



A property that C++ has over other programming languages is that the assignment operation can be used as the rvalue (or part of an rvalue) for another assignment operation. For example:

```
a = 2 + (b = 5);
```

is equivalent to:

```
b = 5;  
a = 2 + b;
```

that means: first assign 5 to variable b and then assign to a the value 2 plus the result of the previous assignment of b (i.e. 5), leaving a with a final value of 7.

The following expression is also valid in C++:

```
a = b = c = 5;
```

It assigns 5 to all the three variables: a, b and c.

Arithmetic operators (+, -, *, /, %)

The five arithmetical operations supported by the C++ language are:

+ addition

- subtraction * multiplication / division % modulo

Operations of addition, subtraction, multiplication and division literally correspond with their respective mathematical operators. The only one that you might not be so used to see is *modulo*; whose operator is the percentage sign (%). Modulo is the operation that gives the remainder of a division of two values. For example, if we write:

```
a = 11 % 3;
```

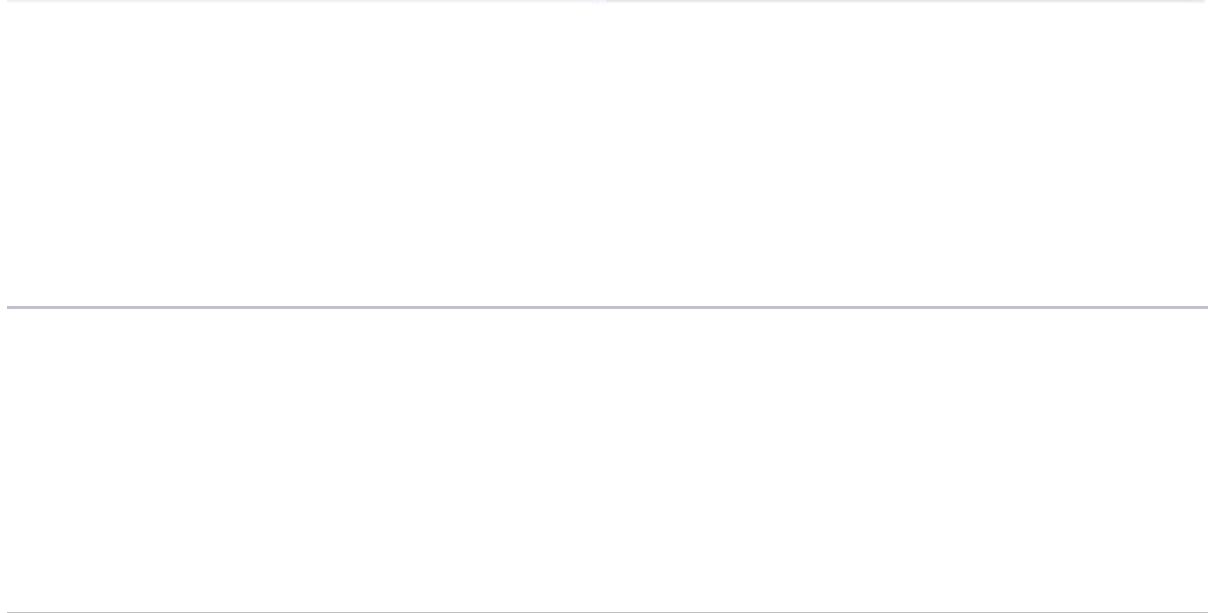
the variable a will contain the value 2, since 2 is the remainder from dividing 11 between 3.

Compound assignment (+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=)

When we want to modify the value of a variable by performing an operation on the value currently stored in that variable we can use compound assignment operators:

expression		is equivalent to
value += increase;	value	= value + increase;
a -= 5;	a = a	- 5;
a /= b;	a = a	/ b;
price *= units + 1;	price	= price * (units + 1);

and the same for all other operators. For example:



// compound assignment operators	5
----------------------------------	---

```
#include <iostream>
using namespace std;

int main ()
{
    int a, b=3;
    a = b;

    a+=2;           // equivalent to a=a+2
    cout << a;
```

```

    return 0;
}

```

Increase and decrease (++ , --)

Shortening even more some expressions, the increase operator (++) and the decrease operator (--) increase or reduce by one the value stored in a variable. They are equivalent to +=1 and to -=1, respectively. Thus:



```
c++;
```

```
c+=1;
```

```
c=c+1;
```

are all equivalent in its functionality: the three of them increase by one the value of c.

In the early C compilers, the three previous expressions probably produced different executable code depending on which one was used. Nowadays, this type of code optimization is generally done automatically by the compiler, thus the three expressions should produce exactly the same executable code.

A characteristic of this operator is that it can be used both as a prefix and as a suffix. That means that it can be written either before the variable identifier (++a) or after it (a++). Although in simple expressions like a++ or ++a both have exactly the same meaning, in other expressions in which the result of the increase or decrease operation is evaluated as a value in an outer expression they may have an important difference in their meaning: In the case that the increase operator is used as a prefix (++a) the value is increased before the result of the expression is evaluated and therefore the increased value is considered in the outer expression; in case that it is used as a suffix (a++) the value stored in a is increased after being evaluated and therefore the value stored before the increase operation is evaluated in the outer expression. Notice the difference:

Example 1	Example 2
B=3;	B=3;
A=++B;	A=B++;
// A contains 4, B contains 4	// A contains 3, B contains 4

In Example 1, B is increased before its value is copied to A. While in Example 2, the value of B is copied to A and then B is increased.

Relational and equality operators (==, !=, >, <, >=, <=)

In order to evaluate a comparison between two expressions we can use the relational and equality operators. The result of a relational operation is a Boolean value that can only be true or false, according to its Boolean result.

We may want to compare two expressions, for example, to know if they are equal or if one is greater than the other is. Here is a list of the relational and equality operators that can be used in C++:



==Equal to

!= Not equal to

Greater than < Less than >=Greater than or equal to <=Less than or equal to

Here there are some examples:

(7 == 5) // evaluates to false.

--

(5 > 4) // evaluates to true.

(3 != 2) // evaluates to true.

(6 >= 6) // evaluates to true.

(5 < 5) // evaluates to false.

Of course, instead of using only numeric constants, we can use any valid expression, including variables. Suppose that a=2, b=3 and c=6,

(a == 5) // evaluates to false since a is not equal to 5.

```
(a*b >= c)    // evaluates to true since (2*3 >= 6) is true.
```

```
(b+4 > a*c)    // evaluates to false since (3+4 > 2*6) is false.
```

```
((b=2) == a) // evaluates to true.
```

Be careful! The operator = (one equal sign) is not the same as the operator == (two equal signs), the first one is an assignment operator (assigns the value at its right to the variable at its left) and the other one (==) is the equality operator that compares whether both expressions in the two sides of it are equal to each other. Thus, in the last expression ((b=2) == a), we first assigned the value 2 to b and then we compared it to a, that also stores the value 2, so the result of the operation is true.

Logical operators (!, &&, ||)

The Operator ! is the C++ operator to perform the Boolean operation NOT, it has only one operand, located at its right, and the only thing that it does is to inverse the value of it, producing false if its operand is true and true if its operand is false. Basically, it returns the opposite Boolean value of evaluating its operand. For example:

!(5 == 5)	// evaluates to false because the expression at its right (5 == 5) is true.
!(6 <= 4)	// evaluates to true because (6 <= 4) would be false.
!true	// evaluates to false
!false	// evaluates to true.

The logical operators && and || are used when evaluating two expressions to obtain a single relational result. The operator && corresponds with Boolean logical operation AND. This operation results true if both its two operands are true, and false otherwise. The following panel shows the result of operator && evaluating the expression a && b:

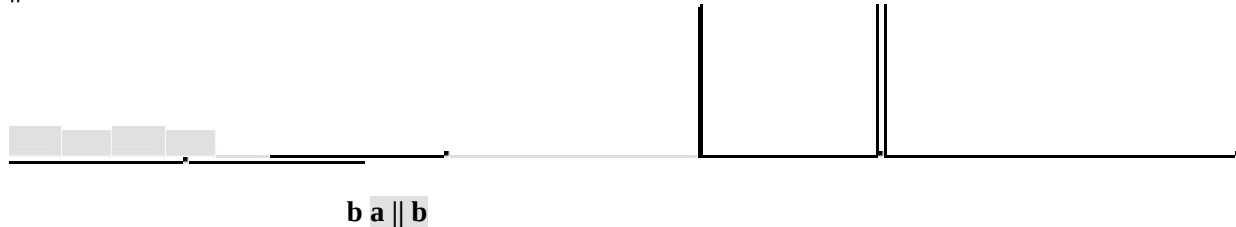


true true true true falsefalse falsetrue false falsefalsefalse

The operator || corresponds with Boolean logical operation OR. This operation results true if either one of its two operands is true, thus being false only when both operands are false themselves. Here are the possible results of a || b:



|| OPERATOR



true true true true false true false true true false false false

For example:

```
(5 == 5) && (3 > 6) // evaluates to false ( true && false ).  
(5 == 5) || (3 > 6) // evaluates to true ( true || false ).
```

Conditional operator (?)

The conditional operator evaluates an expression returning a value if that expression is true and a different one if the expression is evaluated as false. Its format is:

condition ? result1 : result2

If condition is true the expression will return result1, if it is not it will return result2.

```
7==5 ? 4 : 3 // returns 3, since 7 is not equal to 5.
```

```
7==5+2 ? 4 : 3 // returns 4, since 7 is equal to 5+2.
```

```
5>3 ? a : b // returns the value of a, since 5 is greater than 3.
```

```
a>b ? a : b // returns whichever is greater, a or b.
```




// conditional operator

```
#include <iostream>
```

```
using namespace std;
```

```
int main ()  
{
```

```
    int a,b,c;
```

```
    a=2;
```

```
    b=7;
```

```
c = (a>b) ? a : b;  
  
cout << c;  
  
return 0;  
}
```

In this example a was 2 and b was 7, so the expression being evaluated (a>b) was not true, thus the first value specified after the question mark was discarded in favor of the second value (the one after the colon) which was b, with a value of 7.

Comma operator (,)

The comma operator (,) is used to separate two or more expressions that are included where only one expression is expected. When the set of expressions has to be evaluated for a value, only the rightmost expression is considered.

For example, the following code:

```
a = (b=3, b+2);
```



Would first assign the value 3 to b, and then assign b+2 to variable a. So, at the end, variable a would contain the value 5 while variable b would contain value 3.

Bitwise Operators (&, |, ^, ~, <<, >>)

Bitwise operators modify variables considering the bit patterns that represent the values they store.

operator	asm equivalent	description
&	AND	Bitwise AND
	OR	Bitwise Inclusive OR
^	XOR	Bitwise Exclusive OR
~	NOT	Unary complement (bit inversion)
<<	SHL	Shift Left
>>	SHR	Shift Right

Explicit type casting operator

Type casting operators allow you to convert a datum of a given type to another. There are several ways to do this in C++. The simplest one, which has been inherited from the C language, is to precede the expression to be converted by the new type enclosed between parentheses (()):

```
int i;
```

```
float f = 3.14;  
i = (int) f;
```

The previous code converts the float number 3.14 to an integer value (3), the remainder is lost. Here, the typecasting operator was (int). Another way to do the same thing in C++ is using the functional notation: preceding the expression to be converted by the type and enclosing the expression between parentheses:

```
i = int ( f );
```

Both ways of type casting are valid in C++.

sizeof()

This operator accepts one parameter, which can be either a type or a variable itself and returns the size in bytes of that type or object:

```
a = sizeof (char);
```

This will assign the value 1 to a because char is a one-byte long type. The value returned by sizeof is a constant, so it is always determined before program execution.

Other operators

Later in these tutorials, we will see a few more operators, like the ones referring to pointers or the specifics for object-oriented programming. Each one is treated in its respective section.

Precedence of operators

When writing complex expressions with several operands, we may have some doubts about which operand is evaluated first and which later. For example, in this expression:

```
a = 5 + 7 % 2
```



we may doubt if it really means:

```
a = 5 + (7 % 2)    // with a result of 6, or
```

```
a = (5 + 7) % 2    // with a result of 0
```

The correct answer is the first of the two expressions, with a result of 6. There is an established order with the priority of each operator, and not only the arithmetic ones (those whose preference come from mathematics) but for all the operators which can appear in C++. From greatest to lowest priority, the priority order is as follows:

Level	Operator	Description	Grouping
1	::	scope	Left-to-right
2	() [] . -> ++ -- dynamic_cast static_cast reinterpret_cast const_cast typeid	postfix	Left-to-right
	++ -- ~ ! sizeof new delete	unary (prefix)	right
3	* &	indirection and reference	Right-to-left
		(pointers)	left
	+ -	unary sign operator	
4	(type)	type casting	Right-to-left
			left
5	.* ->*	pointer-to-member	Left-to-right
			right
6	* / %	multiplicative	Left-to-right
			right
7	+ -	additive	Left-to-right
			right
8	<< >>	shift	Left-to-right
			right
9	< > <= >=	relational	Left-to-right
			right
10	== !=	equality	Left-to-right

			right
11	&	bitwise AND	Left-to-
			right
12	^	bitwise XOR	Left-to-
			right
13		bitwise OR	Left-to-
			right
14	&&	logical AND	Left-to-
			right
15		logical OR	Left-to-
			right
16	?:	conditional	Right-to-
			left
17	= *= /= %= += -= >>= <<= &= ^= =	assignment	Right-to-
			left
18	,	comma	Left-to-
			right

Grouping defines the precedence order in which operators are evaluated in the case that there are several operators of the same level in an expression.

All these precedence levels for operators can be manipulated or become more legible by removing possible ambiguities using parentheses signs (and), as in this example:

```
a = 5 + 7 % 2;
```



might be written either as:

```
a = 5 + (7 % 2);
```

or

```
a = (5 + 7) % 2;
```

depending on the operation that we want to perform.

So if you want to write complicated expressions and you are not completely sure of the precedence levels, always include parentheses. It will also become a code easier to read.



Basic Input/Output

Until now, the example programs of previous sections provided very little interaction with the user, if any at all. Using the standard input and output library, we will be able to interact with the user by printing messages on the screen and getting the user's input from the keyboard.

C++ uses a convenient abstraction called *streams* to perform input and output operations in sequential media such as the screen or the keyboard. A stream is an object where a program can either insert or extract characters to/from it. We do not really need to care about many specifications about the physical media associated with the stream - we only need to know it will accept or provide characters sequentially.

The standard C++ library includes the header file `iostream`, where the standard input and output stream objects are declared.

Standard Output (`cout`)

By default, the standard output of a program is the screen, and the C++ stream object defined to access it is `cout`.

`cout` is used in conjunction with the *insertion operator*, which is written as `<<` (two "less than" signs).

<code>cout << "Output sentence"; //</code>	<code>prints Output</code>	<code>sentence on screen</code>
<code>cout << 120; //</code>	<code>prints number</code>	<code>120 on screen</code>
<code>cout << x; //</code>	<code>prints the content of x on screen</code>	

The `<<` operator inserts the data that follows it into the stream preceding it. In the examples above it inserted the constant string `Output sentence`, the numerical constant `120` and variable `x` into the standard output stream `cout`. Notice that the sentence in the first instruction is enclosed between double quotes (`"`) because it is a constant string of characters. Whenever we want to use constant strings of characters we must enclose them between double quotes (`"`) so that they can be clearly distinguished from variable names. For example, these two sentences have very different results:

<code>cout</code>	<code><<</code>	<code>"Hello"; //</code>	<code>prints</code>	<code>Hello</code>
<code>cout</code>	<code><<</code>	<code>Hello; //</code>	<code>prints</code>	<code>the content of Hello variable</code>

The insertion operator (`<<`) may be used more than once in a single statement:

```
cout << "Hello, " << "I am " << "a C++ statement";
```

This last statement would print the message `Hello, I am a C++ statement` on the screen. The utility of

repeating the insertion operator (<<) is demonstrated when we want to print out a combination of variables and constants or more than one variable:

```
cout << "Hello, I am " << age << " years old and my zipcode is " << zipcode;
```

If we assume the age variable to contain the value 24 and the zipcode variable to contain 90064 the output of the previous statement would be:

```
Hello, I am 24 years old and my zipcode is 90064
```

It is important to notice that cout does not add a line break after its output unless we explicitly indicate it, therefore, the following statements:



```
cout << "This is a sentence.";
```

```
cout << "This is another sentence.";
```

will be shown on the screen one following the other without any line break between them:

This is a sentence.This is another sentence.

even though we had written them in two different insertions into cout. In order to perform a line break on the output we must explicitly insert a new-line character into cout. In C++ a new-line character can be specified as `\n` (backslash, n):

```
cout << "First sentence.\n ";
```

```
cout << "Second sentence.\nThird sentence.";
```

This produces the following output:

First sentence.
Second sentence.

Third sentence.

Additionally, to add a new-line, you may also use the `endl` manipulator. For example:

```
cout << "First sentence." << endl;
```

```
cout << "Second sentence." << endl;
```

would print out:

First sentence.
Second sentence.

The endl manipulator produces a newline character, exactly as the insertion of '\n' does, but it also has an additional behavior when it is used with buffered streams: the buffer is flushed. Anyway, cout will be an unbuffered stream in most cases, so you can generally use both the \n escape character and the endl manipulator in order to specify a new line without any difference in its behavior.

Standard Input (cin).

The standard input device is usually the keyboard. Handling the standard input in C++ is done by applying the overloaded operator of extraction (>>) on the cin stream. The operator must be followed by the variable that will store the data that is going to be extracted from the stream. For example:

```
int age;  
cin >> age;
```

The first statement declares a variable of type int called age, and the second one waits for an input from cin (the keyboard) in order to store it in this integer variable.

cin can only process the input from the keyboard once the RETURN key has been pressed. Therefore, even if you request a single character, the extraction from cin will not process the input until the user presses RETURN after the character has been introduced.

You must always consider the type of the variable that you are using as a container with cin extractions. If you request an integer you will get an integer, if you request a character you will get a character and if you request a string of characters you will get a string of characters.



// i/o example

```
#include <iostream>
using namespace std;
```

```
int main ()
```

```
{
```

```
    int i;
```

```
    cout << "Please enter an integer value: "; cin >> i;
```

```
    cout << "The value you entered is " << i; cout << " and its double is " << i*2 << ".\n"; return
```

```
    0;  
}
```

Please enter an integer value: 702

The value you entered is 702 and its double is 1404.

The user of a program may be one of the factors that generate errors even in the simplest programs that use cin (like the one we have just seen). Since if you request an integer value and the user introduces a name (which generally is a string of characters), the result may cause your program to misoperate since it is not what we were expecting from the user. So when you use the data input provided by cin extractions you will have to trust that the user of your program will be cooperative and that he/she will not introduce his/her name or something similar when an integer value is requested. A little ahead, when we see the stringstream class we will see a possible solution for the errors that can be caused by this type of user input.

You can also use cin to request more than one datum input from the user:

```
cin >> a >> b;
```

is equivalent to:

```
cin >> a;  
cin >> b;
```

In both cases the user must give two data, one for variable a and another one for variable b that may be separated by any valid blank separator: a space, a tab character or a newline.

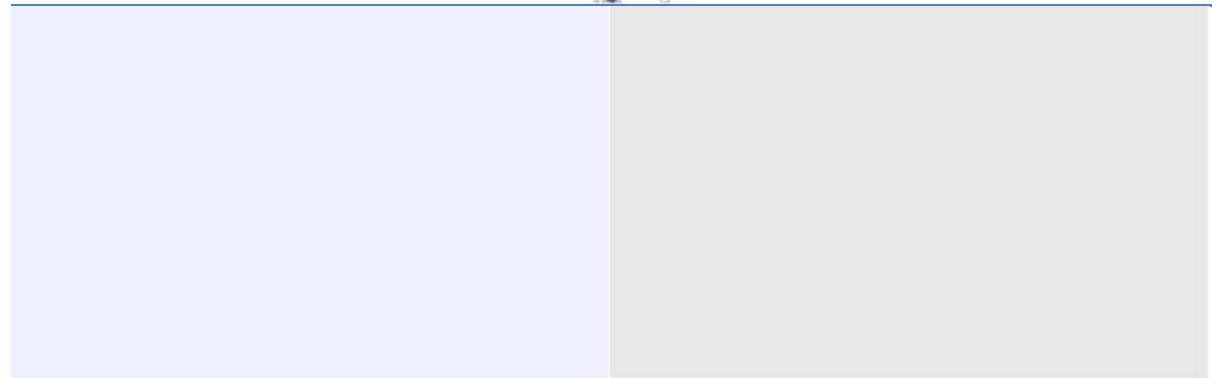
cin and strings

We can use cin to get strings with the extraction operator (>>) as we do with fundamental data type variables:

```
cin >> mystring;
```

However, as it has been said, cin extraction stops reading as soon as it finds any blank space character, so in this case we will be able to get just one word for each extraction. This behavior may or may not be what we want; for example if we want to get a sentence from the user, this extraction operation would not be useful.

In order to get entire lines, we can use the function getline, which is the more recommendable way to get user input with cin:



```
cin with strings #include <iostream> #include <string> using namespace std;

int main ()
{
    string mystr;
    cout << "What's your name? "; getline (cin, mystr);
```



```

    cout << "Hello " << mystr << ".\n"; cout << "What is your favorite team? "; getline (cin,
    mystr);

    cout << "I like " << mystr << " too!\n"; return 0;

}

```

What's your name? Juan Souli Hello Juan Souli.

What is your favorite team? The Isotopes I like The Isotopes too!

Notice how in both calls to getline we used the same string identifier (mystr). What the program does in the second call is simply to replace the previous content by the new one that is introduced.

stringstream

The standard header file <sstream> defines a class called stringstream that allows a string-based object to be treated as a stream. This way we can perform extraction or insertion operations from/to strings, which is especially useful to convert strings to numerical values and vice versa. For example, if we want to extract an integer from a string we can write:

```

// stringstream example
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

int main ()
{
    string mystr ("1204");
    int myint;

    stringstream(mystr) >> myint;
}

```

This declares a string object with a value of "1204", and an int object. Then we use stringstream's constructor to construct an object of this type from the string object. Because we can use stringstream objects as if they were streams, we can extract an integer from it as we would have done on cin by applying the extractor operator (>>) on it followed by a variable of type int.

After this piece of code, the variable myint will contain the numerical value 1204.

// stringstreams	Enter price: 22.25
#include <iostream>	Enter quantity: 7
#include <string>	Total price: 155.75
#include <sstream>	
using namespace std;	
int main ()	

{	
string mystr;	
float price=0;	
int quantity=0;	
cout << "Enter price: ";	
getline (cin,mystr);	
stringstream(mystr) >> price;	
cout << "Enter quantity: ";	
getline (cin,mystr);	
stringstream(mystr) >> quantity;	
cout << "Total price: " << price*quantity <<	
endl;	
return 0;	
}	

In this example, we acquire numeric values from the standard input indirectly. Instead of extracting numeric values directly from the standard input, we get lines from the standard input (cin) into a string object (mystr), and then we extract the integer values from this string into a variable of type int (quantity).



Using this method, instead of direct extractions of integer values, we have more control over what happens with the input of numeric values from the user, since we are separating the process of obtaining input from the user (we now simply ask for lines) with the interpretation of that input. Therefore, this method is usually preferred to get numerical values from the user in all programs that are intensive in user input.



Control Structures

Control Structures

A program is usually not limited to a linear sequence of instructions. During its process it may bifurcate, repeat code or take decisions. For that purpose, C++ provides control structures that serve to specify what has to be done by our program, when and under which circumstances.

With the introduction of control structures we are going to have to introduce a new concept: the *compound-statement* or *block*. A block is a group of statements which are separated by semicolons (;) like all C++ statements, but grouped together in a block enclosed in braces: { }:

```
{ statement1; statement2; statement3; }
```

Most of the control structures that we will see in this section require a generic statement as part of its syntax. A statement can be either a simple statement (a simple instruction ending with a semicolon) or a compound statement (several instructions grouped in a block), like the one just described. In the case that we want the statement to be a simple statement, we do not need to enclose it in braces ({}). But in the case that we want the statement to be a compound statement it must be enclosed between braces ({}), forming a block.

Conditional structure: if and else

The if keyword is used to execute a statement or block only if a condition is fulfilled. Its form is:

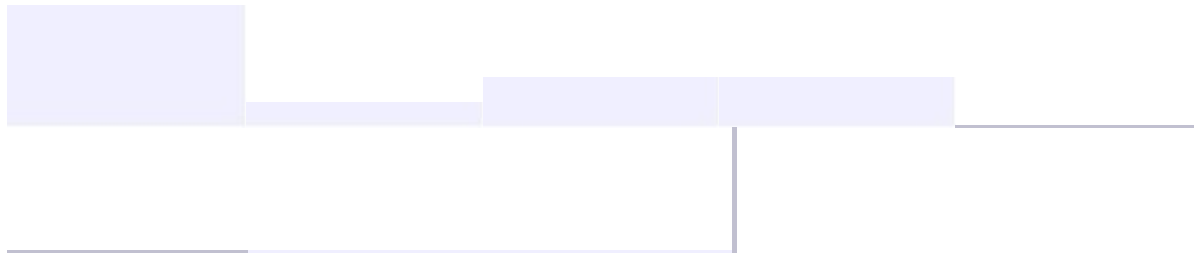
```
if (condition) statement
```

Where condition is the expression that is being evaluated. If this condition is true, statement is executed. If it is false, statement is ignored (not executed) and the program continues right after this conditional structure.

For example, the following code fragment prints x is 100 only if the value stored in the x variable is indeed 100:

```
if (x == 100)
    cout << "x is 100";
```

If we want more than a single statement to be executed in case that the condition is true we can specify a block using braces { }:



```
if (x == 100)
{
    cout << "x is ";
    cout << x;
}
```

We can additionally specify what we want to happen if the condition is not fulfilled by using the keyword `else`. Its form used in conjunction with `if` is:

`if (condition) statement1 else statement2`

For example:



```
if (x == 100)
    cout << "x is 100";
else
    cout << "x is not 100";
```

prints on the screen x is 100 if indeed x has a value of 100, but if it has not -and only if not- it prints out x is not 100.

The if + else structures can be concatenated with the intention of verifying a range of values. The following example shows its use telling if the value currently stored in x is positive, negative or none of them (i.e. zero):

```
if (x > 0)
    cout << "x is positive";
else if (x < 0)
    cout << "x is negative";
else
    cout << "x is 0";
```

Remember that in case that we want more than a single statement to be executed, we must group them in a block by enclosing them in braces { }.

Iteration structures (loops)

Loops have as purpose to repeat a statement a certain number of times or while a condition is fulfilled.

The while loop

Its format is:

while (expression) statement

and its functionality is simply to repeat statement while the condition set in expression is true.

For example, we are going to make a program to countdown using a while-loop:

// custom countdown using while	Enter the starting number > 8
	8, 7, 6, 5, 4, 3, 2, 1, FIRE!
#include <iostream>	
using namespace std;	
int main ()	
{	
int n;	
cout << "Enter the starting number > ";	
cin >> n;	
while (n>0) {	
cout << n << ", ";	
--n;	
}	
cout << "FIRE!\n";	
return 0;	
}	

When the program starts the user is prompted to insert a starting number for the countdown. Then the while loop begins, if the value entered by the user fulfills the condition $n > 0$ (that n is greater than zero) the block that follows the condition will be executed and repeated while the condition ($n > 0$) remains being true.

The whole process of the previous program can be interpreted according to the following script (beginning in main):



User assigns a value to n

The while condition is checked ($n > 0$). At this point there are two possibilities:

condition is true: statement is executed (to step 3)
condition is false: ignore statement and continue after it (to step 5)

Execute statement: `cout << n << " "; --n;`

(prints the value of n on the screen and decreases n by 1)

End of block. Return automatically to step 2

Continue the program right after the block: print FIRE! and end program.

When creating a while-loop, we must always consider that it has to end at some point, therefore we must provide within the block some method to force the condition to become false at some point, otherwise the loop will continue looping forever. In this case we have included `--n;` that decreases the value of the variable that is being evaluated in the condition (n) by one - this will eventually make the condition ($n > 0$) to become false after a certain number of loop iterations: to be more specific, when n becomes 0, that is where our while-loop and our countdown end.

Of course this is such a simple action for our computer that the whole countdown is performed instantly without any practical delay between numbers.

The do-while loop

Its format is:

`do statement while (condition);`

Its functionality is exactly the same as the while loop, except that condition in the do-while loop is evaluated after the execution of statement instead of before, granting at least one execution of statement even if condition is never fulfilled. For example, the following example program echoes any number you enter until you enter 0.

<code>// number echoer</code>	Enter number (0 to end): 12345
	You entered: 12345
<code>#include <iostream></code>	Enter number (0 to end): 160277
<code>using namespace std;</code>	You entered: 160277

	Enter number (0 to end): 0
int main ()	You entered: 0
{	
unsigned long n;	
do {	
cout << "Enter number (0 to end): ";	
cin >> n;	
cout << "You entered: " << n << "\n";	
} while (n != 0);	
return 0;	
}	

The do-while loop is usually used when the condition that has to determine the end of the loop is determined within the loop statement itself, like in the previous case, where the user input within the block is what is used to determine if the loop has to end. In fact if you never enter the value 0 in the previous example you can be prompted for more numbers forever.

The for loop

Its format is:

for (initialization; condition; increase) statement;



and its main function is to repeat statement while condition remains true, like the while loop. But in addition, the for loop provides specific locations to contain an initialization statement and an increase statement. So this loop is specially designed to perform a repetitive action with a counter which is initialized and increased on each iteration.

It works in the following way:

initialization is executed. Generally it is an initial value setting for a counter variable. This is executed only once.

condition is checked. If it is true the loop continues, otherwise the loop ends and statement is skipped (not executed).

statement is executed. As usual, it can be either a single statement or a block enclosed in braces { }.

finally, whatever is specified in the increase field is executed and the loop gets back to step 2.

Here is an example of countdown using a for loop:	
// countdown using a for loop	10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!
#include <iostream>	
using namespace std;	
int main ()	
{	
for (int n=10; n>0; n--) {	
cout << n << ", ";	
}	
cout << "FIRE!\n";	
return 0;	
}	

The initialization and increase fields are optional. They can remain empty, but in all cases the semicolon signs between them must be written. For example we could write: for (;n<10;) if we wanted to specify no initialization and no increase; or for (;n<10;n++) if we wanted to include an increase field but no initialization (maybe because the variable was already initialized before).

Optionally, using the comma operator (,) we can specify more than one expression in any of the fields

included in a for loop, like in initialization, for example. The comma operator (,) is an expression separator, it serves to separate more than one expression where only one is generally expected. For example, suppose that we wanted to initialize more than one variable in our loop:

```

for ( n=0, i=100 ; n!=i ; n++, i-- )
{
    // whatever here...
}
```

This loop will execute for 50 times if neither n or i are modified within the loop:

```

for ( n=0, i=100 ; n!=i ; n++, i-- )
    ↑
    Initialization
    Condition
    Increase
```

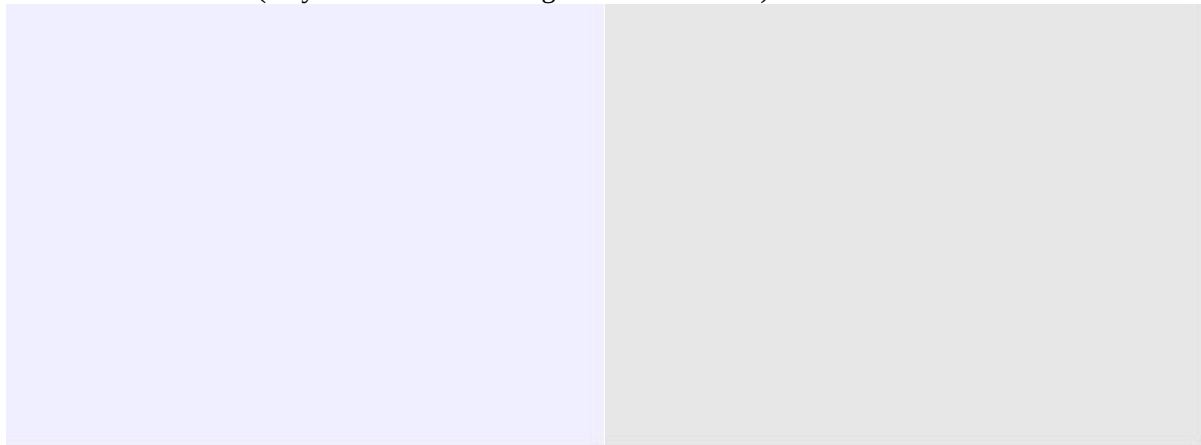
n starts with a value of 0, and i with 100, the condition is $n \neq i$ (that n is not equal to i). Because n is increased by one and i decreased by one, the loop's condition will become false after the 50th loop, when both n and i will be equal to 50.



Jump statements.

The break statement

Using break we can leave a loop even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end. For example, we are going to stop the count down before its natural end (maybe because of an engine check failure?):



// break loop example	10, 9, 8, 7, 6, 5, 4, 3, countdown aborted!
-----------------------	---

```
#include <iostream>
using namespace std;

int main ()
{
    int n;

    for (n=10; n>0; n--)
    {
        cout << n << " ";

        if (n==3)
        {
            cout << "countdown aborted!";
            break;
        }
    }

    return 0;
}
```

The continue statement

The continue statement causes the program to skip the rest of the loop in the current iteration as if the end of the statement block had been reached, causing it to jump to the start of the following iteration. For example, we are going to skip the number 5 in our countdown:

// continue loop example	10, 9, 8, 7, 6, 4, 3, 2, 1, FIRE!
#include <iostream>	
using namespace std;	

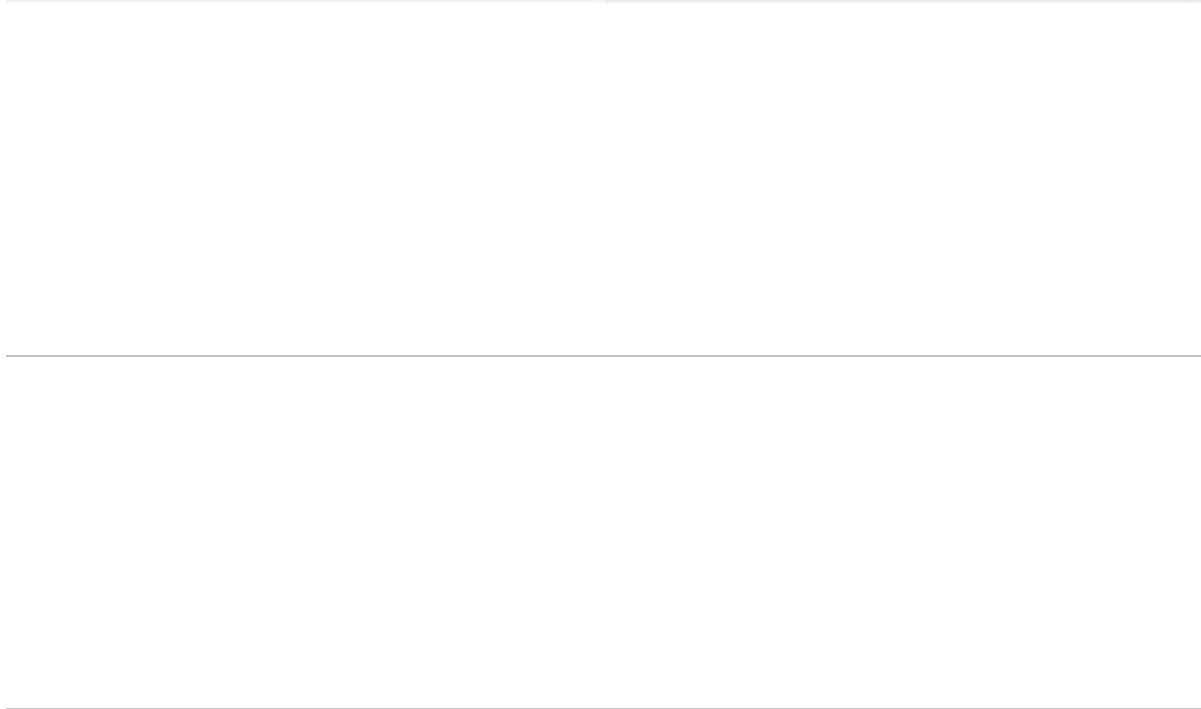
<code>int main ()</code>	
<code>{</code>	
<code>for (int n=10; n>0; n--) {</code>	
<code>if (n==5) continue;</code>	
<code>cout << n << ", ";</code>	
<code>}</code>	
<code>cout << "FIRE!\n";</code>	
<code>return 0;</code>	
<code>}</code>	

The goto statement

goto allows to make an absolute jump to another point in the program. You should use this feature with caution since its execution causes an unconditional jump ignoring any type of nesting limitations.

The destination point is identified by a label, which is then used as an argument for the goto statement. A label is made of a valid identifier followed by a colon (:).

Generally speaking, this instruction has no concrete use in structured or object oriented programming aside from those that low-level programming fans may find for it. For example, here is our countdown loop using goto:



<code>// goto loop example</code>	10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!
-----------------------------------	--------------------------------------

```
#include <iostream>
using namespace std;
```

```
int main ()
```

```
{
```

```
    int n=10;
    loop:
```



```

        cout << n << ", ";
        n--;

        if (n>0) goto loop;
        cout << "FIRE!\n";

        return 0;
}

```

The exit function

exit is a function defined in the cstdlib library.

The purpose of exit is to terminate the current program with a specific exit code. Its prototype is:

```
void exit (int exitcode);
```

The exitcode is used by some operating systems and may be used by calling programs. By convention, an exit code of 0 means that the program finished normally and any other value means that some error or unexpected results happened.

The selective structure: switch.

The syntax of the switch statement is a bit peculiar. Its objective is to check several possible constant values for an expression. Something similar to what we did at the beginning of this section with the concatenation of several if and else if instructions. Its form is the following:

```

switch (expression)
{
    case constant1:
        group of statements 1;
        break;

    case constant2:
        group of statements 2;
        break;

    .
    .
    .

    default:
        default group of statements
}

```

It works in the following way: switch evaluates expression and checks if it is equivalent to constant1, if it is,

it executes group of statements 1 until it finds the break statement. When it finds this break statement the program jumps to the end of the switch selective structure.

If expression was not equal to constant1 it will be checked against constant2. If it is equal to this, it will execute group of statements 2 until a break keyword is found, and then will jump to the end of the switch selective structure.

Finally, if the value of expression did not match any of the previously specified constants (you can include as many case labels as values you want to check), the program will execute the statements included after the default: label, if it exists (since it is optional).

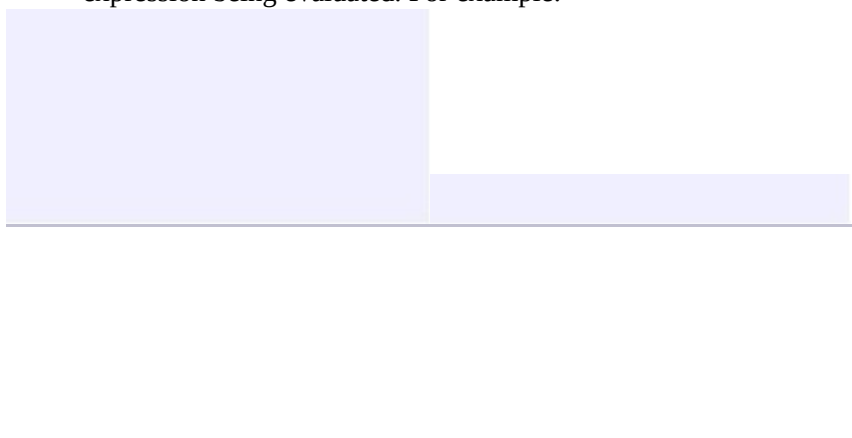





Both of the following code fragments have the same behavior:

switch example	if-else equivalent
switch (x) {	if (x == 1) {
case 1:	cout << "x is 1";
cout << "x is 1";	}
break;	else if (x == 2) {
case 2:	cout << "x is 2";
cout << "x is 2";	}
break;	else {
default:	cout << "value of x unknown";
cout << "value of x unknown";	}
}	

The switch statement is a bit peculiar within the C++ language because it uses labels instead of blocks. This forces us to put break statements after the group of statements that we want to be executed for a specific condition. Otherwise the remainder statements -including those corresponding to other labels- will also be executed until the end of the switch selective block or a break statement is reached.

For example, if we did not include a break statement after the first group for case one, the program will not automatically jump to the end of the switch selective block and it would continue executing the rest of statements until it reaches either a break instruction or the end of the switch selective block. This makes unnecessary to include braces { } surrounding the statements for each of the cases, and it can also be useful to execute the same block of instructions for different possible values for the expression being evaluated. For example:



```
switch (x) {  
    case 1:  
          
    case 2:  
          
    case 3:  
          
        cout << "x is 1, 2 or 3";  
        break;  
    default:  
        cout << "x is not 1, 2 nor 3";  
}
```

Notice that switch can only be used to compare an expression against constants. Therefore we cannot put variables as labels (for example case n: where n is a variable) or ranges (case (1..3):) because they are not valid C++ constants.

If you need to check ranges or values that are not constants, use a concatenation of if and else if statements.



Functions (I)

Using functions we can structure our programs in a more modular way, accessing all the potential that structured programming can offer to us in C++.

A function is a group of statements that is executed when it is called from some point of the program. The following is its format:

```
type name ( parameter1, parameter2, ...) { statements }
```

where:

type is the data type specifier of the data returned by the function.

name is the identifier by which it will be possible to call the function.

parameters (as many as needed): Each parameter consists of a data type specifier followed by an

identifier, like any regular variable declaration (for example: int x) and which acts within the function as a regular local variable. They allow to pass arguments to the function when it is called. The different parameters are separated by commas.

statements is the function's body. It is a block of statements surrounded by braces { }.

Here you have the first function example:	
// function example	The result is 8
#include <iostream>	
using namespace std;	
int addition (int a, int b)	
{	
int r;	
r=a+b;	
return (r);	

}	
int main ()	
{	
int z;	
z = addition (5,3);	
cout << "The result is " << z;	
return 0;	
}	

In order to examine this code, first of all remember something said at the beginning of this tutorial: a C++ program always begins its execution by the main function. So we will begin there.

We can see how the main function begins by declaring the variable z of type int. Right after that, we see a call to a function called addition. Paying attention we will be able to see the similarity between the structure of the call to the function and the declaration of the function itself some code lines above:

`int addition (int a, int b)` ↑ ↑ `z = addition (5 , 3);`

The parameters and arguments have a clear correspondence. Within the main function we called to addition passing two values: 5 and 3, that correspond to the int a and int b parameters declared for function addition.



At the point at which the function is called from within main, the control is lost by main and passed to function addition. The value of both arguments passed in the call (5 and 3) are copied to the local variables int a and int b within the function.

Function addition declares another local variable (int r), and by means of the expression $r = a + b$, it assigns to r the result of a plus b. Because the actual parameters passed for a and b are 5 and 3 respectively, the result is 8.

The following line of code:

```
return (r);
```

finalizes function addition, and returns the control back to the function that called it in the first place (in this case, main). At this moment the program follows its regular course from the same point at which it was interrupted by the call to addition. But additionally, because the return statement in function addition specified a value: the content of variable r (return (r);), which at that moment had a value of 8. This value becomes the value of evaluating the function call.

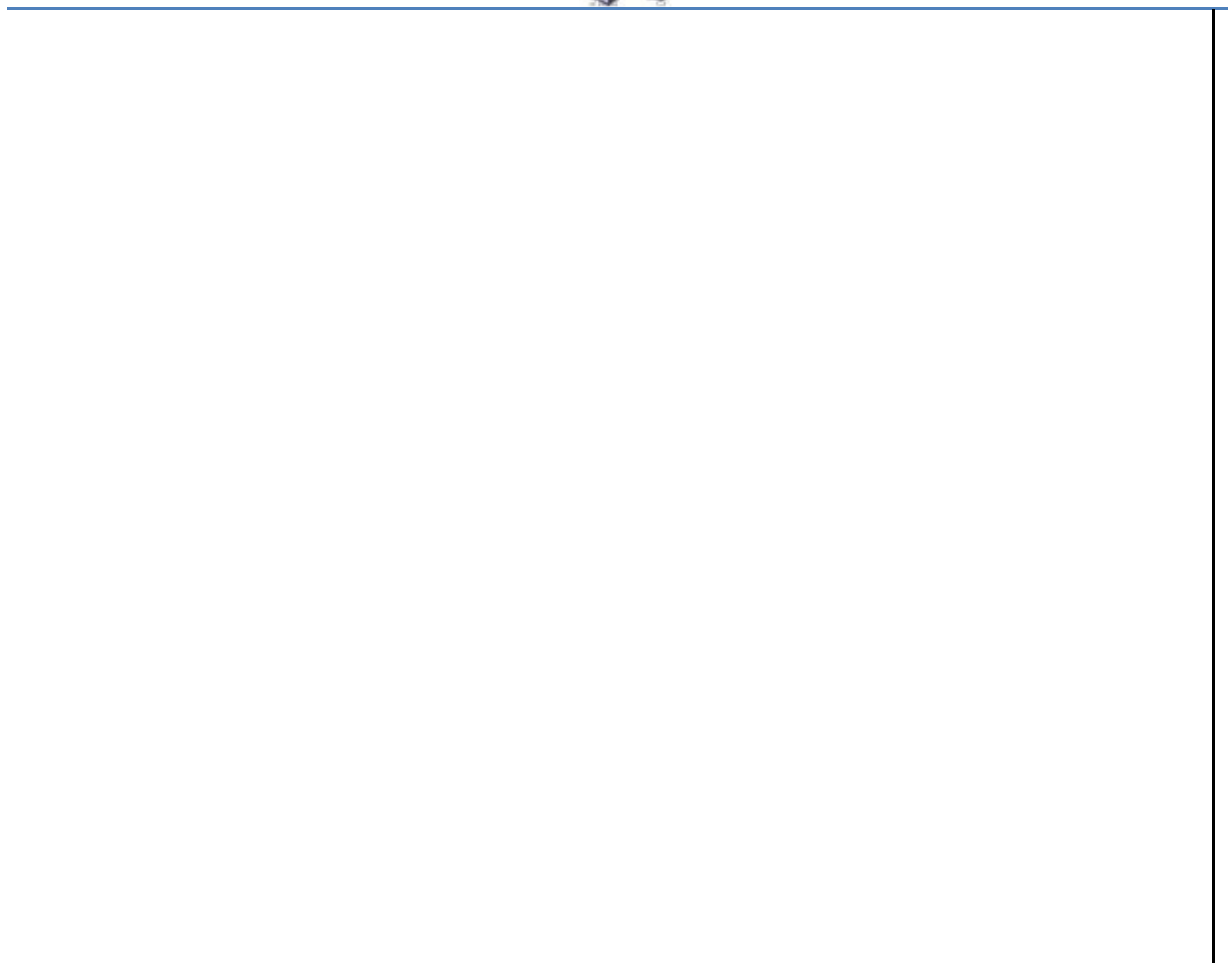
```
int addition (int a, int b) {  
    // ...  
    return 8;  
}  
int z = addition ( 5 , 3 );
```

So being the value returned by a function the value given to the function call itself when it is evaluated, the variable z will be set to the value returned by addition (5, 3), that is 8. To explain it another way, you can imagine that the call to a function (addition (5,3)) is literally replaced by the value it returns (8).

The following line of code in main is:

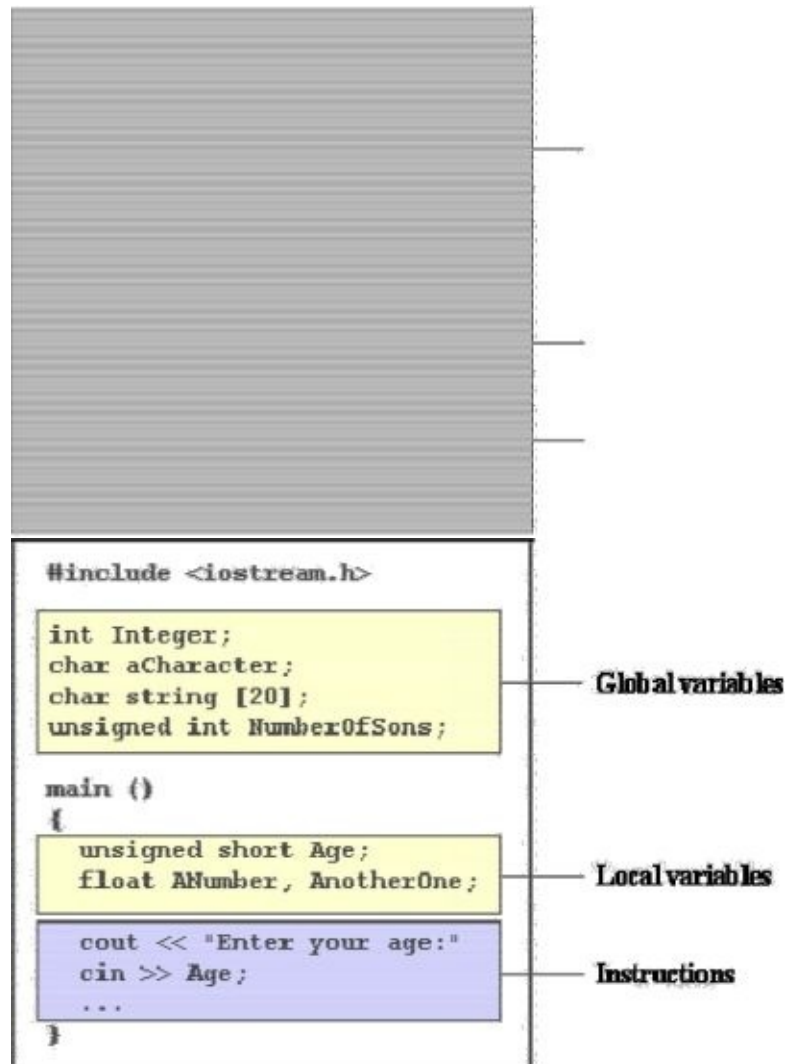
```
cout << "The result is " << z;
```

That, as you may already expect, produces the printing of the result on the screen.



Scope of variables

The scope of variables declared within a function or any other inner block is only their own function or their own block and cannot be used outside of them. For example, in the previous example it would have been impossible to use the variables a, b or r directly in function main since they were variables local to function addition. Also, it would have been impossible to use the variable z directly within function addition, since this was a variable local to the function main.



Therefore, the scope of local variables is limited to the same block level in which they are declared. Nevertheless, we also have the possibility to declare global variables; These are visible from any point of the code, inside and outside all functions. In order to declare global variables you simply have to declare the variable outside any function or block; that means, directly in the body of the program.

And here is another example about functions:



// function example	The first result is 5
#include <iostream>	The second result is 5
using namespace std;	The third result is 2
	The fourth result is 6
int subtraction (int a, int b)	
{	
int r;	
r=a-b;	
return (r);	
}	
int main ()	
{	
int x=5, y=3, z;	
z = subtraction (7,2);	
cout << "The first result is " << z << '\n';	
cout << "The second result is " << subtraction (7,2) << '\n';	
cout << "The third result is " << subtraction (x,y) << '\n';	
z= 4 + subtraction (x,y);	
cout << "The fourth result is " << z << '\n';	
return 0;	
}	

In this case we have created a function called subtraction. The only thing that this function does is to subtract both passed parameters and to return the result.

Nevertheless, if we examine function main we will see that we have made several calls to function subtraction. We have used some different calling methods so that you see other ways or moments when a function can be called.

In order to fully understand these examples you must consider once again that a call to a function could be replaced by the value that the function call itself is going to return. For example, the first case (that you should already know because it is the same pattern that we have used in previous examples):

```
z = subtraction (7,2);
```

```
cout << "The first result is " << z;
```

If we replace the function call by the value it returns (i.e., 5), we would have:

```
z = 5;
```

```
cout << "The first result is " << z;
```

As well as

```
cout << "The second result is " << subtraction (7,2);
```

has the same result as the previous call, but in this case we made the call to subtraction directly as an insertion parameter for cout. Simply consider that the result is the same as if we had written:

```
cout << "The second result is " << 5;
```

since 5 is the value returned by subtraction (7,2).

In the case of:

```
cout << "The third result is " << subtraction (x,y);
```



The only new thing that we introduced is that the parameters of subtraction are variables instead of constants. That is perfectly valid. In this case the values passed to function subtraction are the values of x and y, that are 5 and 3 respectively, giving 2 as result.

The fourth case is more of the same. Simply note that instead of:

```
z = 4 + subtraction (x,y);
```

we could have written:

```
z = subtraction (x,y) + 4;
```

with exactly the same result. I have switched places so you can see that the semicolon sign (;) goes at the end of the whole statement. It does not necessarily have to go right after the function call. The explanation might be once again that you imagine that a function can be replaced by its returned value:

```
z = 4 + 2;
```

```
z = 2 + 4;
```

Functions with no type. The use of void.

If you remember the syntax of a function declaration:

type name (argument1, argument2 ...) statement

you will see that the declaration begins with a type, that is the type of the function itself (i.e., the type of the datum that will be returned by the function with the return statement). But what if we want to return no value?

Imagine that we want to make a function just to show a message on the screen. We do not need it to return any value. In this case we should use the void type specifier for the function. This is a special specifier that indicates absence of type.

// void function example	I'm a function!
#include <iostream>	
using namespace std;	
void printmessage ()	

{	
cout << "I'm a function!";	
}	
int main ()	
{	
printmessage ();	
return 0;	
}	

void can also be used in the function's parameter list to explicitly specify that we want the function to take no actual parameters when it is called. For example, function printmessage could have been declared as:


```
void printmessage (void)
{
    cout << "I'm a function!";
}
```

Although it is optional to specify void in the parameter list. In C++, a parameter list can simply be left blank if we want a function with no parameters.



What you must always remember is that the format for calling a function includes specifying its name and enclosing its parameters between parentheses. The non-existence of parameters does not exempt us from the obligation to write the parentheses. For that reason the call to `printmessage` is:

```
printmessage ();
```

The parentheses clearly indicate that this is a call to a function and not the name of a variable or some other C++ statement. The following call would have been incorrect:

```
printmessage;
```




Functions (II)

Arguments passed by value and by reference.

Until now, in all the functions we have seen, the arguments passed to the functions have been passed *by value*. This means that when calling a function with parameters, what we have passed to the function were copies of their values but never the variables themselves. For example, suppose that we called our first function addition using the following code:

```
int x=5, y=3, z;  
  
z = addition ( x , y );
```

What we did in this case was to call to function addition passing the values of x and y, i.e. 5 and 3 respectively, but not the variables x and y themselves.

```
int addition (int a, int b) {  
    ↑  
    z = addition ( 5 , 3 );  
}
```

This way, when the function addition is called, the value of its local variables a and b become 5 and 3 respectively, but any modification to either a or b within the function addition will not have any effect in the values of x and y outside it, because variables x and y were not themselves passed to the function, but only copies of their values at the moment the function was called.

But there might be some cases where you need to manipulate from inside a function the value of an external variable. For that purpose we can use arguments passed by reference, as in the function duplicate of the following example:

// passing parameters by reference	x=2, y=6, z=14
#include <iostream>	
using namespace std;	
void duplicate (int& a, int& b, int& c)	
{	

a*=2;	
b*=2;	
c*=2;	
}	
int main ()	
{	
int x=1, y=3, z=7;	
duplicate (x, y, z);	
cout << "x=" << x << ", y=" << y << ", z=" << z;	
return 0;	
}	

The first thing that should call your attention is that in the declaration of duplicate the type of each parameter was followed by an ampersand sign (&). This ampersand is what specifies that their corresponding arguments are to be passed *by reference* instead of *by value*.

When a variable is passed by reference we are not passing a copy of its value, but we are somehow passing the variable itself to the function and any modification that we do to the local variables will have an effect in their counterpart variables passed as arguments in the call to the function.



```
void duplicate (int& a,int& b,int& c)
duplicate ( x , y , z );
```



To explain it in another way, we associate a, b and c with the arguments passed on the function call (x, y and z) and any change that we do on a within the function will affect the value of x outside it. Any change that we do on b will affect y, and the same with c and z.

That is why our program's output, that shows the values stored in x, y and z after the call to duplicate, shows the values of all the three variables of main doubled.

If when declaring the following function:

```
void duplicate (int& a, int& b, int& c)
```

we had declared it this way:

```
void duplicate (int a, int b, int c)
```

i.e., without the ampersand signs (&), we would have not passed the variables by reference, but a copy of their values instead, and therefore, the output on screen of our program would have been the values of x, y and z without having been modified.

Passing by reference is also an effective way to allow a function to return more than one value. For example, here is a function that returns the previous and next numbers of the first parameter passed.

// more than one returning value	Previous=99, Next=101
#include <iostream>	
using namespace std;	
void prevnext (int x, int& prev, int& next)	
{	
prev = x-1;	
next = x+1;	
}	

<code>int main ()</code>	
<code>{</code>	
<code>int x=100, y, z;</code>	
<code>prevnext (x, y, z);</code>	
<code>cout << "Previous=" << y << ", Next=" << z;</code>	
<code>return 0;</code>	
<code>}</code>	

Default values in parameters.

When declaring a function we can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function. To do that, we simply have to use the assignment operator and a value for the arguments in the function declaration. If a value for that parameter is not passed when the function is called, the default value is used, but if a value is specified this default value is ignored and the passed value is used instead. For example:



// default values in functions	6
#include <iostream>	5
using namespace std;	
int divide (int a, int b=2)	
{	
int r;	
r=a/b;	
return (r);	
}	
int main ()	
{	
cout << divide (12);	
cout << endl;	
cout << divide (20,4);	
return 0;	
}	

As we can see in the body of the program there are two calls to function divide. In the first one:

divide (12)

we have only specified one argument, but the function divide allows up to two. So the function divide has assumed that the second parameter is 2 since that is what we have specified to happen if this parameter was not passed (notice the function declaration, which finishes with int b=2, not just int b). Therefore the result of this function call is 6(12/2).

In the second call:

divide (20,4)

there are two parameters, so the default value for b(int b=2) is ignored and b takes the value passed as argument, that is 4, making the result returned equal to 5(20/4).



Overloaded functions.

In C++ two different functions can have the same name if their parameter types or number are different. That means that you can give the same name to more than one function if they have either a different number of parameters or different types in their parameters. For example:

// overloaded function	10
#include <iostream>	2.5
using namespace std;	
int operate (int a, int b)	
{	
return (a*b);	
}	
float operate (float a, float b)	
{	
return (a/b);	
}	
int main ()	
{	
int x=5,y=2;	
float n=5.0,m=2.0;	
cout << operate (x,y);	
cout << "\n";	
cout << operate (n,m);	
cout << "\n";	
return 0;	
}	

In this case we have defined two functions with the same name, operate, but one of them accepts two parameters of type int and the other one accepts them of type float. The compiler knows which one to call in each case by examining the types passed as arguments when the function is called. If it is called with two ints as its arguments it calls to the function that has two int parameters in its prototype and if it is called with two floats it will call to the one which has two float parameters in its prototype.

In the first call to operate the two arguments passed are of type int, therefore, the function with the first

prototype is called; This function returns the result of multiplying both parameters. While the second call passes two arguments of type float, so the function with the second prototype is called. This one has a different behavior: it divides one parameter by the other. So the behavior of a call to operate depends on the type of the arguments passed because the function has been *overloaded*.

Notice that a function cannot be overloaded only by its return type. At least one of its parameters must have a different type.

inline functions.

The inline specifier indicates the compiler that inline substitution is preferred to the usual function call mechanism for a specific function. This does not change the behavior of a function itself, but is used to suggest to the compiler that the code generated by the function body is inserted at each point the function is called, instead of being inserted only once and perform a regular call to it, which generally involves some additional overhead in running time.

The format for its declaration is:

```
inline type name ( arguments ... ) { instructions ... }
```

and the call is just like the call to any other function. You do not have to include the inline keyword when calling the function, only in its declaration.



Most compilers already optimize code to generate inline functions when it is more convenient. This specifier only indicates the compiler that inline is preferred for this function.

Recursivity.

Recursivity is the property that functions have to be called by themselves. It is useful for many tasks, like sorting or calculate the factorial of numbers. For example, to obtain the factorial of a number (n!) the mathematical formula would be:

$$n! = n * (n-1) * (n-2) * (n-3) \dots * 1$$

more concretely, 5! (factorial of 5) would be:	
$5! = 5 * 4 * 3 * 2 * 1 = 120$	
and a recursive function to calculate this in C++ could be:	
// factorial calculator	Please type a number: 9
#include <iostream>	9! = 362880
using namespace std;	
long factorial (long a)	
{	
if (a > 1)	
return (a * factorial (a-1));	
else	
return (1);	
}	
int main ()	
{	
long number;	
cout << "Please type a number: ";	
cin >> number;	
cout << number << "! = " << factorial (number);	
return 0;	
}	

Notice how in function factorial we included a call to itself, but only if the argument passed was greater than 1, since otherwise the function would perform an infinite recursive loop in which once it arrived to 0 it would continue multiplying by all the negative numbers (probably provoking a stack overflow error on runtime).

This function has a limitation because of the data type we used in its design (long) for more simplicity. The results given will not be valid for values much greater than 10! or 15!, depending on the system you compile it.

Declaring functions.

Until now, we have defined all of the functions before the first appearance of calls to them in the source code. These calls were generally in function main which we have always left at the end of the source code. If you try to repeat some of the examples of functions described so far, but placing the function main before any of the other functions that were called from within it, you will most likely obtain compiling errors. The reason is that to be able to call a function it must have been declared in some earlier point of the code, like we have done in all our examples.

But there is an alternative way to avoid writing the whole code of a function before it can be used in main or in some other function. This can be achieved by declaring just a prototype of the function before it is used, instead of the entire definition. This declaration is shorter than the entire definition, but significant enough for the compiler to determine its return type and the types of its parameters.

Its form is:



type name (argument_type1, argument_type2, ...);

It is identical to a function definition, except that it does not include the body of the function itself (i.e., the function statements that in normal definitions are enclosed in braces { }) and instead of that we end the prototype declaration with a mandatory semicolon (;).

The parameter enumeration does not need to include the identifiers, but only the type specifiers. The inclusion of a name for each parameter as in the function definition is optional in the prototype declaration. For example, we can declare a function called `protofunction` with two `int` parameters with any of the following declarations:

```
int protofunction (int first, int second);  
int protofunction (int, int);
```

Anyway, including a name for each variable makes the prototype more legible.

// declaring functions prototypes	Type a number (0 to exit): 9
#include <iostream>	Number is odd.
using namespace std;	Type a number (0 to exit): 6
	Number is even.
void odd (int a);	Type a number (0 to exit): 1030
void even (int a);	Number is even.
	Type a number (0 to exit): 0
int main ()	Number is even.
{	
int i;	
do {	
cout << "Type a number (0 to exit): ";	
cin >> i;	
odd (i);	
} while (i!=0);	
return 0;	
}	
void odd (int a)	
{	

<code>if ((a%2)!=0) cout << "Number is odd.\n";</code>	
<code>else even (a);</code>	
<code>}</code>	
<code>void even (int a)</code>	
<code>{</code>	
<code>if ((a%2)==0) cout << "Number is even.\n";</code>	
<code>else odd (a);</code>	
<code>}</code>	

This example is indeed not an example of efficiency. I am sure that at this point you can already make a program with the same result, but using only half of the code lines that have been used in this example. Anyway this example illustrates how prototyping works. Moreover, in this concrete example the prototyping of at least one of the two functions is necessary in order to compile the code without errors.

The first things that we see are the declaration of functions odd and even:

```
void odd (int a);
void even (int a);
```

This allows these functions to be used before they are defined, for example, in main, which now is located where some people find it to be a more logical place for the start of a program: the beginning of the source code.

Anyway, the reason why this program needs at least one of the functions to be declared before it is defined is because in odd there is a call to even and in even there is a call to odd. If none of the two functions had been



previously declared, a compilation error would happen, since either odd would not be visible from even (because it has still not been declared), or even would not be visible from odd (for the same reason).

Having the prototype of all functions together in the same place within the source code is found practical by some programmers, and this can be easily achieved by declaring all functions prototypes at the beginning of a program.



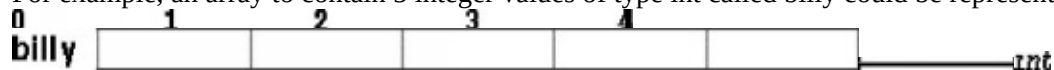
Compound data types

Arrays

An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier.

That means that, for example, we can store 5 values of type `int` in an array without having to declare 5 different variables, each one with a different identifier. Instead of that, using an array we can store 5 different values of the same type, `int` for example, with a unique identifier.

For example, an array to contain 5 integer values of type `int` called `billy` could be represented like this:



where each blank panel represents an element of the array, that in this case are integer values of type `int`. These elements are numbered from 0 to 4 since in arrays the first index is always 0, independently of its length.

Like a regular variable, an array must be declared before it is used. A typical declaration for an array in C++ is:

type name [elements];

where type is a valid type (like `int`, `float`...), name is a valid identifier and the elements field (which is always enclosed in square brackets `[]`), specifies how many of these elements the array has to contain.

Therefore, in order to declare an array called `billy` as the one shown in the above diagram it is as simple as:

```
int billy [5];
```

NOTE: The elements field within brackets `[]` which represents the number of elements the array is going to hold, must be a constant value, since arrays are blocks of non-dynamic memory whose size must be determined before execution. In order to create arrays with a variable length dynamic memory is needed, which is explained later in these tutorials.

Initializing arrays.

When declaring a regular array of local scope (within a function, for example), if we do not specify otherwise, its elements will not be initialized to any value by default, so their content will be undetermined until we store some value in them. The elements of global and static arrays, on the other hand, are automatically initialized with their default values, which for all fundamental types this means they are filled with zeros.

In both cases, local and global, when we declare an array, we have the possibility to assign initial values to each one of its elements by enclosing the values in braces { }. For example:

```
int billy [5] = { 16, 2, 77, 40, 12071 };
```

This declaration would have created an array like this:



The amount of values between braces { } must not be larger than the number of elements that we declare for the array between square brackets []. For example, in the example of array billy we have declared that it has 5 elements and in the list of initial values within braces { } we have specified 5 values, one for each element.

When an initialization of values is provided for an array, C++ allows the possibility of leaving the square brackets empty []. In this case, the compiler will assume a size for the array that matches the number of values included between braces { }:

```
int billy [] = { 16, 2, 77, 40, 12071 };
```

After this declaration, array billy would be 5 ints long, since we have provided 5 initialization values.

Accessing the values of an array.

In any point of a program in which an array is visible, we can access the value of any of its elements individually as if it was a normal variable, thus being able to both read and modify its value. The format is as simple as:

name[index]

Following the previous examples in which billy had 5 elements and each of those elements was of type int, the name which we can use to refer to each element is the following:

billy[0]	billy[1]	billy[2]	billy[3]	billy[4]
billy				

For example, to store the value 75 in the third element of billy, we could write the following statement:

```
billy[2] = 75;
```

and, for example, to pass the value of the third element of billy to a variable called a, we could write:

```
a = billy[2];
```

Therefore, the expression `billy[2]` is for all purposes like a variable of type `int`.

Notice that the third element of `billy` is specified `billy[2]`, since the first one is `billy[0]`, the second one is `billy[1]`, and therefore, the third one is `billy[2]`. By this same reason, its last element is `billy[4]`. Therefore, if we write `billy[5]`, we would be accessing the sixth element of `billy` and therefore exceeding the size of the array.

In C++ it is syntactically correct to exceed the valid range of indices for an array. This can create problems, since accessing out-of-range elements do not cause compilation errors but can cause runtime errors. The reason why this is allowed will be seen further ahead when we begin to use pointers.

At this point it is important to be able to clearly distinguish between the two uses that brackets `[]` have related to arrays. They perform two different tasks: one is to specify the size of arrays when they are declared; and the second one is to specify indices for concrete array elements. Do not confuse these two possible uses of brackets `[]` with arrays.



<code>int billy[5];</code>	<code>//</code>	declaration of a new	array
<code>billy[2] = 75;</code>	<code>//</code>	access to an element	of the array.

If you read carefully, you will see that a type specifier always precedes a variable or array declaration, while it never precedes an access.

Some other valid operations with arrays:	
<code>billy[0] = a;</code>	
<code>billy[a] = 75;</code>	
<code>b = billy [a+2];</code>	
<code>billy[billy[a]] = billy[2] + 5;</code>	
<code>// arrays example</code>	12206
<code>#include <iostream></code>	
<code>using namespace std;</code>	
<code>int billy [] = { 16, 2, 77, 40, 12071};</code>	
<code>int n, result=0;</code>	
<code>int main ()</code>	
<code>{</code>	
<code>for (n=0 ; n<5 ; n++)</code>	
<code>{</code>	
<code>result += billy[n];</code>	
<code>}</code>	
<code>cout << result;</code>	
<code>return 0;</code>	
<code>}</code>	

Multidimensional arrays

Multidimensional arrays can be described as "arrays of arrays". For example, a bidimensional array can be imagined as a bidimensional table made of elements, all of them of a same uniform data type.

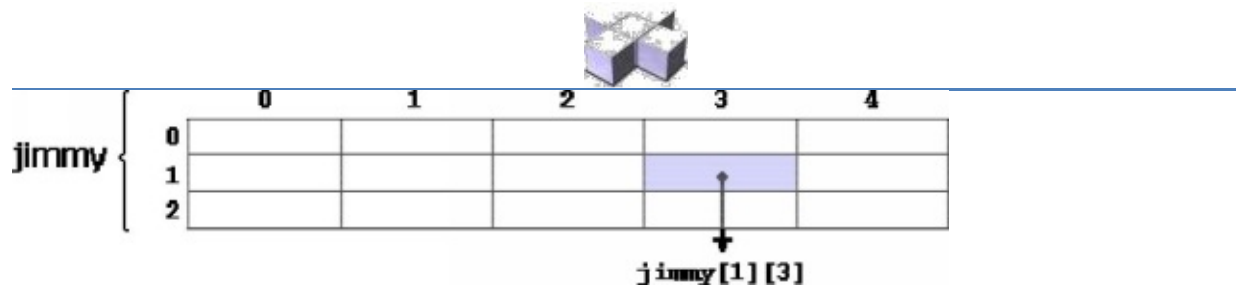
		0	1	2	3	4
jimmy {	0					
	1					
	2					

jimmy represents a bidimensional array of 3 per 5 elements of type int. The way to declare this array in C++ would be:

```
int jimmy [3][5];
```

and, for example, the way to reference the second element vertically and fourth horizontally in an expression would be:

```
jimmy[1][3]
```



(remember that array indices always begin by zero).

Multidimensional arrays are not limited to two indices (i.e., two dimensions). They can contain as many indices as needed. But be careful! The amount of memory needed for an array rapidly increases with each dimension. For example:

```
char century [100][365][24][60][60];
```

declares an array with a char element for each second in a century, that is more than 3 billion chars. So this declaration would consume more than 3 gigabytes of memory!

Multidimensional arrays are just an abstraction for programmers, since we can obtain the same results with a simple array just by putting a factor between its indices:

int	jimmy	[3][5];	//	is	equivalent to
int	jimmy	[15];	//	(3	* 5 = 15)

With the only difference that with multidimensional arrays the compiler remembers the depth of each imaginary dimension for us. Take as example these two pieces of code, with both exactly the same result. One uses a bidimensional array and the other one uses a simple array:

multidimensional array	pseudo-multidimensional array
#define WIDTH 5	#define WIDTH 5
#define HEIGHT 3	#define HEIGHT 3
int jimmy [HEIGHT][WIDTH];	int jimmy [HEIGHT * WIDTH];
int n,m;	int n,m;

int main ()	int main ()
{	{
for (n=0;n<HEIGHT;n++)	for (n=0;n<HEIGHT;n++)
for (m=0;m<WIDTH;m++)	for (m=0;m<WIDTH;m++)
{	{
jimmy[n][m]=(n+1)*(m+1);	jimmy[n*WIDTH+m]=(n+1)*(m+1);
}	}
return 0;	return 0;
}	}

None of the two source codes above produce any output on the screen, but both assign values to the memory block called jimmy in the following way:

		0	1	2	3	4
jimmy {	0	1	2	3	4	5
	1	2	4	6	8	10
	2	3	6	9	12	15



We have used "defined constants" (#define) to simplify possible future modifications of the program. For example, in case that we decided to enlarge the array to a height of 4 instead of 3 it could be done simply by changing the line:

```
#define HEIGHT 3
```

to:

```
#define HEIGHT 4
```

with no need to make any other modifications to the program.

Arrays as parameters

At some moment we may need to pass an array to a function as a parameter. In C++ it is not possible to pass a complete block of memory by value as a parameter to a function, but we are allowed to pass its address. In practice this has almost the same effect and it is a much faster and more efficient operation.

In order to accept arrays as parameters the only thing that we have to do when declaring the function is to specify in its parameters the element type of the array, an identifier and a pair of void brackets []. For example, the following function:

```
void procedure (int arg[])
```

accepts a parameter of type "array of int" called arg. In order to pass to this function an array declared as:

<pre>int myarray [40];</pre>	
it would be enough to write a call like this:	
<pre>procedure (myarray);</pre>	
Here you have a complete example:	
<pre>// arrays as parameters</pre>	5 10 15
<pre>#include <iostream></pre>	2 4 6 8 10
<pre>using namespace std;</pre>	

void printarray (int arg[], int length) {	
for (int n=0; n<length; n++)	
{	
cout << arg[n] << " ";	
cout << "\n";	
}	
int main ()	
{	
int firstarray[] = {5, 10, 15};	
int secondarray[] = {2, 4, 6, 8, 10};	
printarray (firstarray,3);	
printarray (secondarray,5);	
return 0;	
}	

As you can see, the first parameter (int arg[]) accepts any array whose elements are of type int, whatever its length. For that reason we have included a second parameter that tells the function the length of each array that



we pass to it as its first parameter. This allows the for loop that prints out the array to know the range to iterate in the passed array without going out of range.

In a function declaration it is also possible to include multidimensional arrays. The format for a tridimensional array parameter is:

```
base_type[][depth][depth]
```

for example, a function with a multidimensional array as argument could be:

```
void procedure (int myarray[][3][4])
```

Notice that the first brackets [] are left blank while the following ones are not. This is so because the compiler must be able to determine within the function which is the depth of each additional dimension.

Arrays, both simple or multidimensional, passed as function parameters are a quite common source of errors for novice programmers. I recommend the reading of the chapter about Pointers for a better understanding on how arrays operate.



Character Sequences

As you may already know, the C++ Standard Library implements a powerful [string](#) class, which is very useful to handle and manipulate strings of characters. However, because strings are in fact sequences of characters, we can represent them also as plain arrays of char elements.

For example, the following array:

```
char jenny [20];
```

is an array that can store up to 20 elements of type char. It can be represented as:

jenny																			
--------------	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

Therefore, in this array, in theory, we can store sequences of characters up to 20 characters long. But we can also store shorter sequences. For example, jenny could store at some point in a program either the sequence "Hello" or the sequence "Merry christmas", since both are shorter than 20 characters.

Therefore, since the array of characters can store shorter sequences than its total length, a special character is used to signal the end of the valid sequence: the *null character*, whose literal constant can be written as `'\0'` (backslash, zero).

Our array of 20 elements of type char, called jenny, can be represented storing the characters sequences "Hello" and "Merry Christmas" as:

jenny	H	e	l	l	o	\0													
M	e	r	r	y		C	h	r	i	s	t	m	a	s	\0				

Notice how after the valid content a null character (`'\0'`) has been included in order to indicate the end of the

sequence. The panels in gray color represent char elements with undetermined values.

Initialization of null-terminated character sequences

Because arrays of characters are ordinary arrays they follow all their same rules. For example, if we want to initialize an array of characters with some predetermined sequence of characters we can do it just like any other array:

```
char myword[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

In this case we would have declared an array of 6 elements of type char initialized with the characters that form the word "Hello" plus a null character '\0' at the end.

But arrays of char elements have an additional method to initialize their values: using string literals.

In the expressions we have used in some examples in previous chapters, constants that represent entire strings of characters have already showed up several times. These are specified enclosing the text to become a string literal between double quotes ("). For example:

```
"the result is: "
```



is a constant string literal that we have probably used already.

Double quoted strings (") are literal constants whose type is in fact a null-terminated array of characters. So string literals enclosed between double quotes always have a null character ('\0') automatically appended at the end.

Therefore we can initialize the array of char elements called myword with a null-terminated sequence of characters by either one of these two methods:

```
char myword [] = { 'H', 'e', 'l', 'l', 'o', '\0' }; char myword [] = "Hello";
```

In both cases the array of characters myword is declared with a size of 6 elements of type char: the 5 characters that compose the word "Hello" plus a final null character ('\0') which specifies the end of the sequence and that, in the second case, when using double quotes (") it is appended automatically.

Please notice that we are talking about initializing an array of characters in the moment it is being declared, and not about assigning values to them once they have already been declared. In fact because this type of null-terminated arrays of characters are regular arrays we have the same restrictions that we have with any other array, so we are not able to copy blocks of data with an assignment operation.

Assuming mystext is a char[] variable, expressions within a source code like:

```
mystext = "Hello";
```

```
mystext[] = "Hello";
```

would not be valid, like neither would be:

```
mystext = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

The reason for this may become more comprehensible once you know a bit more about pointers, since then it will be clarified that an array is in fact a constant pointer pointing to a block of memory.

Using null-terminated sequences of characters

Null-terminated sequences of characters are the natural way of treating strings in C++, so they can be used as such in many procedures. In fact, regular string literals have this type (`char[]`) and can also be used in most cases.

For example, `cin` and `cout` support null-terminated sequences as valid containers for sequences of characters, so they can be used directly to extract strings of characters from `cin` or to insert them into `cout`. For example:



// null-terminated sequences of characters	Please, enter your first name: John
#include <iostream>	Hello, John!
using namespace std;	
int main ()	
{	
char question[] = "Please, enter your first	
name: ";	
char greeting[] = "Hello, ";	
char yourname [80];	
cout << question;	
cin >> yourname;	
cout << greeting << yourname << "!";	
return 0;	
}	

As you can see, we have declared three arrays of char elements. The first two were initialized with string literal constants, while the third one was left uninitialized. In any case, we have to specify the size of the array: in the first two (question and greeting) the size was implicitly defined by the length of the literal constant they were initialized to. While for yourname we have explicitly specified that it has a size of 80 chars.

Finally, sequences of characters stored in char arrays can easily be converted into string objects just by using the assignment operator:



```
string mystring;
```

```
char myntcs[]="some text";
```

```
mystring = myntcs;
```




Pointers

We have already seen how variables are seen as memory cells that can be accessed using their identifiers. This way we did not have to care about the physical location of our data within memory, we simply used its identifier whenever we wanted to refer to our variable.

The memory of your computer can be imagined as a succession of memory cells, each one of the minimal size that computers manage (one byte). These single-byte memory cells are numbered in a consecutive way, so as, within any block of memory, every cell has the same number as the previous one plus one.

This way, each cell can be easily located in the memory because it has a unique address and all the memory cells follow a successive pattern. For example, if we are looking for cell 1776 we know that it is going to be right between cells 1775 and 1777, exactly one thousand cells after 776 and exactly one thousand cells before cell 2776.

Reference operator (&)

As soon as we declare a variable, the amount of memory needed is assigned for it at a specific location in memory (its memory address). We generally do not actively decide the exact location of the variable within the panel of cells that we have imagined the memory to be - Fortunately, that is a task automatically performed by the operating system during runtime. However, in some cases we may be interested in knowing the address where our variable is being stored during runtime in order to operate with relative positions to it.

The address that locates a variable within memory is what we call a *reference* to that variable. This reference to a variable can be obtained by preceding the identifier of a variable with an ampersand sign (&), known as reference operator, and which can be literally translated as "address of". For example:

```
ted = &andy;
```

This would assign to ted the address of variable andy, since when preceding the name of the variable andy with the reference operator (&) we are no longer talking about the content of the variable itself, but about its reference (i.e., its address in memory).

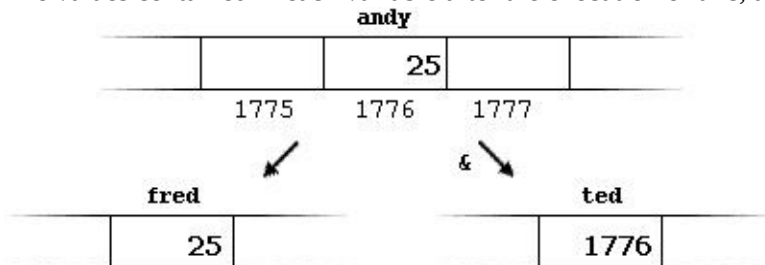
From now on we are going to assume that andy is placed during runtime in the memory address 1776. This number (1776) is just an arbitrary assumption we are inventing right now in order to help clarify some concepts in this tutorial, but in reality, we cannot know before runtime the real value the address of a variable will have in memory.

Consider the following code fragment:

```
int x = 10;
int y = 20;
```

```
andy = 25;  
fred = andy;  
  
ted = &andy;
```

The values contained in each variable after the execution of this, are shown in the following diagram:



First, we have assigned the value 25 to andy (a variable whose address in memory we have assumed to be 1776).



The second statement copied to fred the content of variable andy (which is 25). This is a standard assignment operation, as we have done so many times before.

Finally, the third statement copies to ted not the value contained in andy but a reference to it (i.e., its address, which we have assumed to be 1776). The reason is that in this third assignment operation we have preceded the identifier andy with the reference operator (&), so we were no longer referring to the value of andy but to its reference (its address in memory).

The variable that stores the reference to another variable (like ted in the previous example) is what we call a *pointer*. Pointers are a very powerful feature of the C++ language that has many uses in advanced programming. Farther ahead, we will see how this type of variable is used and declared.

Dereference operator (*)

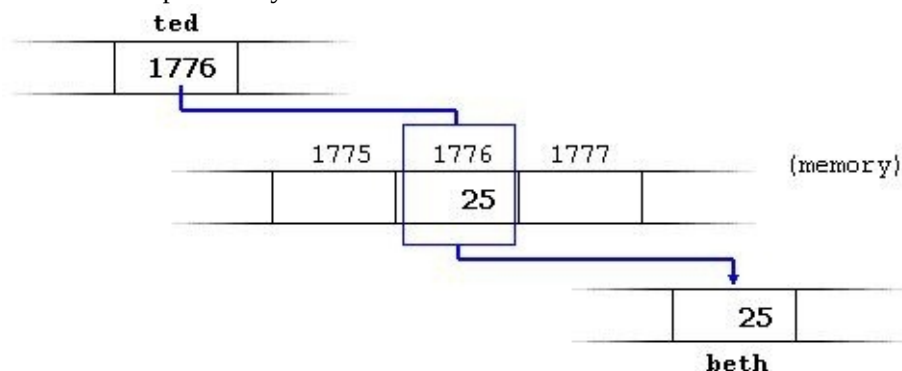
We have just seen that a variable which stores a reference to another variable is called a pointer. Pointers are said to "point to" the variable whose reference they store.

Using a pointer we can directly access the value stored in the variable which it points to. To do this, we simply have to precede the pointer's identifier with an asterisk (*), which acts as dereference operator and that can be literally translated to "value pointed by".

Therefore, following with the values of the previous example, if we write:

```
beth = *ted;
```

(that we could read as: "beth equal to value pointed by ted") beth would take the value 25, since ted is 1776, and the value pointed by 1776 is 25.



You must clearly differentiate that the expression `ted` refers to the value 1776, while `*ted` (with an asterisk `*` preceding the identifier) refers to the value stored at address 1776, which in this case is 25. Notice the difference of including or not including the dereference operator (I have included an explanatory commentary of how each of these two expressions could be read):

<code>beth</code>	<code>=</code>	<code>ted;</code>	<code>//</code>	<code>beth</code>	<code>equal</code>	<code>to</code>	<code>ted (</code>	<code>1776)</code>
<code>beth</code>	<code>=</code>	<code>*ted;</code>	<code>//</code>	<code>beth</code>	<code>equal</code>	<code>to</code>	<code>value</code>	<code>pointed by ted (</code> <code>25)</code>

Notice the difference between the reference and dereference operators:

`&` is the reference operator and can be read as "address of"

`*` is the dereference operator and can be read as "value pointed by"



Thus, they have complementary (or opposite) meanings. A variable referenced with & can be dereferenced with *.

Earlier we performed the following two assignment operations:

```
andy = 25;  
ted = &andy;
```

Right after these two statements, all of the following expressions would give true as result:

```
andy == 25  
&andy == 1776  
  
ted == 1776  
*ted == 25
```

The first expression is quite clear considering that the assignment operation performed on andy was andy=25. The second one uses the reference operator (&), which returns the address of variable andy, which we assumed it to have a value of 1776. The third one is somewhat obvious since the second expression was true and the assignment operation performed on ted was ted=&andy. The fourth expression uses the dereference operator (*) that, as we have just seen, can be read as "value pointed by", and the value pointed by ted is indeed 25.

So, after all that, you may also infer that for as long as the address pointed by ted remains unchanged the following expression will also be true:

```
*ted == andy
```

Declaring variables of pointer types

Due to the ability of a pointer to directly refer to the value that it points to, it becomes necessary to specify in its declaration which data type a pointer is going to point to. It is not the same thing to point to a char as to point to an int or a float.

The declaration of pointers follows this format:

type * name;

where type is the data type of the value that the pointer is intended to point to. This type is not the type of the pointer itself! but the type of the data the pointer points to. For example:



int * number;

char * character;

float * greatnumber;

These are three declarations of pointers. Each one is intended to point to a different data type, but in fact all of them are pointers and all of them will occupy the same amount of space in memory (the size in memory of a pointer depends on the platform where the code is going to run). Nevertheless, the data to which they point to do not occupy the same amount of space nor are of the same type: the first one points to an int, the second one to a char and the last one to a float. Therefore, although these three example variables are all of them pointers which occupy the same size in memory, they are said to have different types: int*, char* and float* respectively, depending on the type they point to.

I want to emphasize that the asterisk sign (*) that we use when declaring a pointer only means that it is a pointer (it is part of its type compound specifier), and should not be confused with the dereference operator that we have seen a bit earlier, but which is also written with an asterisk (*). They are simply two different things represented with the same sign.



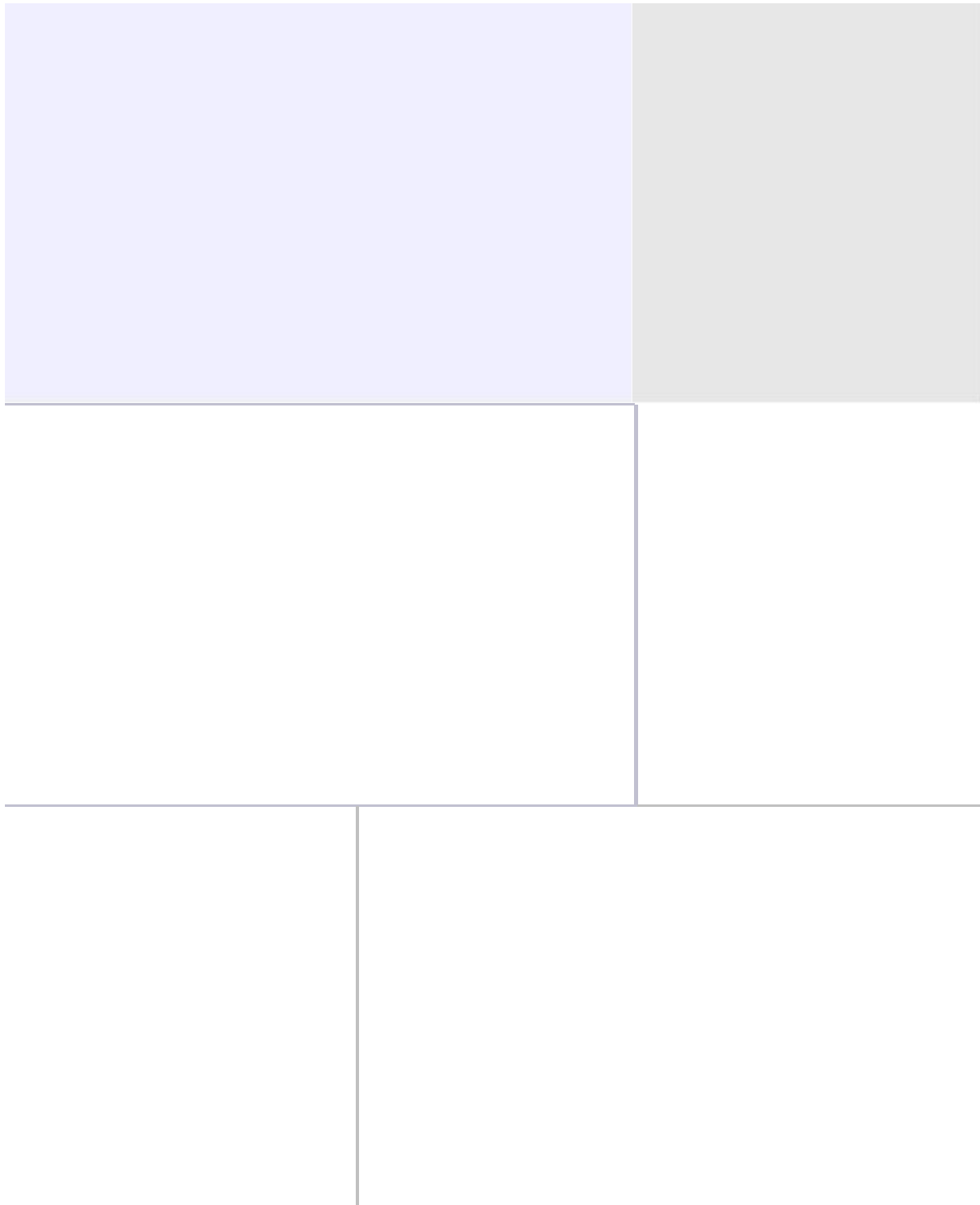
Now have a look at this code:		
// my first pointer		firstvalue is 10
#include <iostream>		secondvalue is 20
using namespace std;		
int main ()		
{		
int firstvalue, secondvalue;		
int * mypointer;		
mypointer = &firstvalue;		
*mypointer = 10;		
mypointer = &secondvalue;		
*mypointer = 20;		
cout << "firstvalue is " << firstvalue << endl;		
cout << "secondvalue is " << secondvalue << endl;		
return 0;		
}		

Notice that even though we have never directly set a value to either firstvalue or secondvalue, both end up with a value set indirectly through the use of mypointer. This is the procedure:

First, we have assigned as value of mypointer a reference to firstvalue using the reference operator (&). And then we have assigned the value 10 to the memory location pointed by mypointer, that because at this moment is pointing to the memory location of firstvalue, this in fact modifies the value of firstvalue.

In order to demonstrate that a pointer may take several different values during the same program I have repeated the process with secondvalue and that same pointer, mypointer.

Here is an example a little bit more elaborated:



```
// more pointers
```

```
#include <iostream>  
using namespace std;
```

```
int main ()
```

firstvalue is 10

secondvalue is 20

```

{

    int firstvalue = 5, secondvalue = 15;
    int * p1, * p2;

    p1 = &firstvalue;    // p1 = address of firstvalue

    p2 = &secondvalue; // p2 = address of secondvalue
    *p1 = 10;            // value pointed by p1 = 10

    *p2 = *p1;           // value pointed by p2 = value pointed by
p1

    p1 = p2;             // p1 = p2 (value of pointer is copied)
    *p1 = 20;            // value pointed by p1 = 20

    cout << "firstvalue is " << firstvalue << endl; cout << "secondvalue is " << secondvalue <<
    endl; return 0;

}

```

I have included as a comment on each line how the code can be read: ampersand (&) as "address of" and asterisk (*) as "value pointed by".

Notice that there are expressions with pointers p1 and p2, both with and without dereference operator (*). The meaning of an expression using the dereference operator (*) is very different from one that does not: When this operator precedes the pointer name, the expression refers to the value being pointed, while when a pointer name appears without this operator, it refers to the value of the pointer itself (i.e. the address of what the pointer is pointing to).



Another thing that may call your attention is the line:

```
int * p1, * p2;
```

This declares the two pointers used in the previous example. But notice that there is an asterisk (*) for each pointer, in order for both to have type `int*` (pointer to `int`).

Otherwise, the type for the second variable declared in that line would have been `int` (and not `int*`) because of precedence relationships. If we had written:

```
int * p1, p2;
```

`p1` would indeed have `int*` type, but `p2` would have type `int` (spaces do not matter at all for this purpose). This is due to operator precedence rules. But anyway, simply remembering that you have to put one asterisk per pointer is enough for most pointer users.

Pointers and arrays

The concept of array is very much bound to the one of pointer. In fact, the identifier of an array is equivalent to the address of its first element, as a pointer is equivalent to the address of the first element that it points to, so in fact they are the same concept. For example, supposing these two declarations:

```
int numbers [20];  
int * p;
```

The following assignment operation would be valid:

```
p = numbers;
```

After that, `p` and `numbers` would be equivalent and would have the same properties. The only difference is that we could change the value of pointer `p` by another one, whereas `numbers` will always point to the first of the 20 elements of type `int` with which it was defined. Therefore, unlike `p`, which is an ordinary pointer, `numbers` is an array, and an array can be considered a *constant pointer*. Therefore, the following allocation would not be valid:

```
numbers = p;
```

Because `numbers` is an array, so it operates as a constant pointer, and we cannot assign values to constants.

Due to the characteristics of variables, all expressions that include pointers in the following example are perfectly valid:



// more pointers		10, 20, 30, 40, 50,
#include <iostream>		
using namespace std;		
int main ()		
{		
int numbers[5];		
int * p;		
p = numbers;	*p = 10;	
p++; *p = 20;		
p = &numbers[2];	*p = 30;	
p = numbers + 3;	*p = 40;	
p = numbers;	*(p+4) = 50;	
for (int n=0; n<5; n++)		
cout << numbers[n] << ", ";		
return 0;		
}		

In the chapter about arrays we used brackets ([]) several times in order to specify the index of an element of the array to which we wanted to refer. Well, these bracket sign operators [] are also a dereference operator known as *offset operator*. They dereference the variable they follow just as * does, but they also add the number between brackets to the address being dereferenced. For example:

a[5] = 0; // a [offset of 5] = 0

*(a+5) = 0; // pointed by (a+5) = 0

These two expressions are equivalent and valid both if a is a pointer or if a is an array.

Pointer initialization

When declaring pointers we may want to explicitly specify which variable we want them to point to:


```
int number;  
  
int *tommy = &number;
```

The behavior of this code is equivalent to:

```
int number;  
int *tommy;  
  
tommy = &number;
```

When a pointer initialization takes place we are always assigning the reference value to where the pointer points

(tommy), never the value being pointed (*tommy). You must consider that at the moment of declaring a pointer, the asterisk (*) indicates only that it is a pointer, it is not the dereference operator (although both use the same sign: *). Remember, they are two different functions of one sign. Thus, we must take care not to confuse the previous code with:

```
int number;  
int *tommy;  
  
*tommy = &number;
```

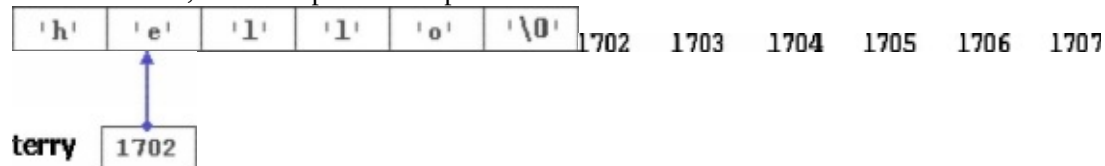
that is incorrect, and anyway would not have much sense in this case if you think about it.

As in the case of arrays, the compiler allows the special case that we want to initialize the content at which the pointer points with constants at the same moment the pointer is declared:

```
char * terry = "hello";
```

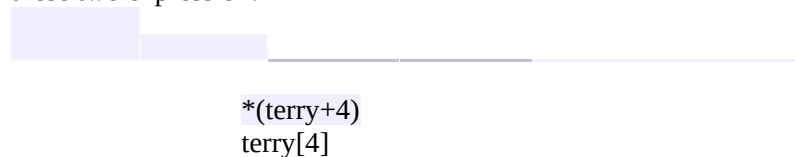



In this case, memory space is reserved to contain "hello" and then a pointer to the first character of this memory block is assigned to terry. If we imagine that "hello" is stored at the memory locations that start at addresses 1702, we can represent the previous declaration as:



It is important to indicate that terry contains the value 1702, and not 'h' nor "hello", although 1702 indeed is the address of both of these.

The pointer terry points to a sequence of characters and can be read as if it was an array (remember that an array is just like a constant pointer). For example, we can access the fifth element of the array with any of these two expression:



Both expressions have a value of 'o' (the fifth element of the array).

Pointer arithmetics

To conduct arithmetical operations on pointers is a little different than to conduct them on regular integer data types. To begin with, only addition and subtraction operations are allowed to be conducted with them, the others make no sense in the world of pointers. But both addition and subtraction have a different behavior with pointers according to the size of the data type to which they point.

When we saw the different fundamental data types, we saw that some occupy more or less space than others in the memory. For example, let's assume that in a given compiler for a specific machine, char takes 1 byte, short takes 2 bytes and long takes 4.

Suppose that we define three pointers in this compiler:



```
char *mychar;
```

```
short *myshort;
```

```
long *mylong;
```

and that we know that they point to memory locations 1000, 2000 and 3000 respectively.

So if we write:

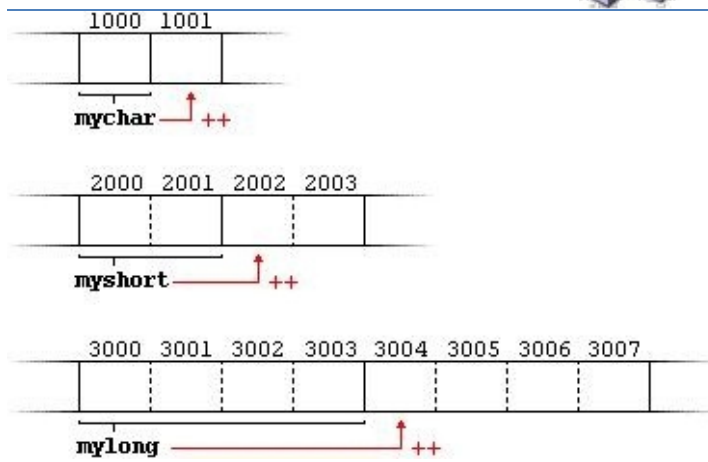


```
mychar++;
```

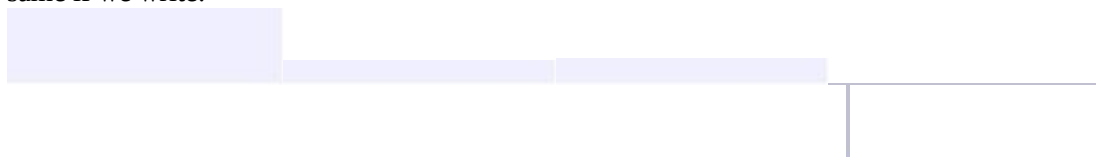
```
myshort++;
```

```
mylong++;
```

mychar, as you may expect, would contain the value 1001. But not so obviously, myshort would contain the value 2002, and mylong would contain 3004, even though they have each been increased only once. The reason is that when adding one to a pointer we are making it to point to the following element of the same type with which it has been defined, and therefore the size in bytes of the type pointed to is added to the pointer.



This is applicable both when adding and subtracting any number to a pointer. It would happen exactly the same if we write:



```
mychar = mychar + 1;  
myshort = myshort + 1;  
  
mylong = mylong + 1;
```

Both the increase (++) and decrease (--) operators have greater operator precedence than the dereference operator (*), but both have a special behavior when used as suffix (the expression is evaluated with the value it had before being increased). Therefore, the following expression may lead to confusion:

```
*p++
```

Because ++ has greater precedence than *, this expression is equivalent to *(p++). Therefore, what it does is to increase the value of p (so it now points to the next element), but because ++ is used as postfix the whole expression is evaluated as the value pointed by the original reference (the address the pointer pointed to before being increased).

Notice the difference with:

```
(*p)++
```

Here, the expression would have been evaluated as the value pointed by p increased by one. The value of p (the pointer itself) would not be modified (what is being modified is what it is being pointed to by this pointer).

If we write:

```
*p++ = *q++;
```

Because ++ has a higher precedence than *, both p and q are increased, but because both increase operators (++) are used as postfix and not prefix, the value assigned to *p is *q before both p and q are increased. And then both are increased. It would be roughly equivalent to:

```
*p = *q;  
++p;  
  
++q;
```

Like always, I recommend you to use parentheses () in order to avoid unexpected results and to give more legibility to the code.

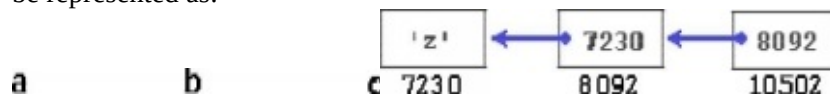


Pointers to pointers

C++ allows the use of pointers that point to pointers, that these, in its turn, point to data (or even to other pointers). In order to do that, we only need to add an asterisk (*) for each level of reference in their declarations:

```
char a;  
char * b;  
  
char ** c;  
a = 'z';  
  
b = &a;  
c = &b;
```

This, supposing the randomly chosen memory locations for each variable of 7230, 8092 and 10502, could be represented as:



The value of each variable is written inside each cell; under the cells are their respective addresses in memory.

The new thing in this example is variable c, which can be used in three different levels of indirection, each one of them would correspond to a different value:

c has type char** and a value of 8092

c has type char and a value of 7230

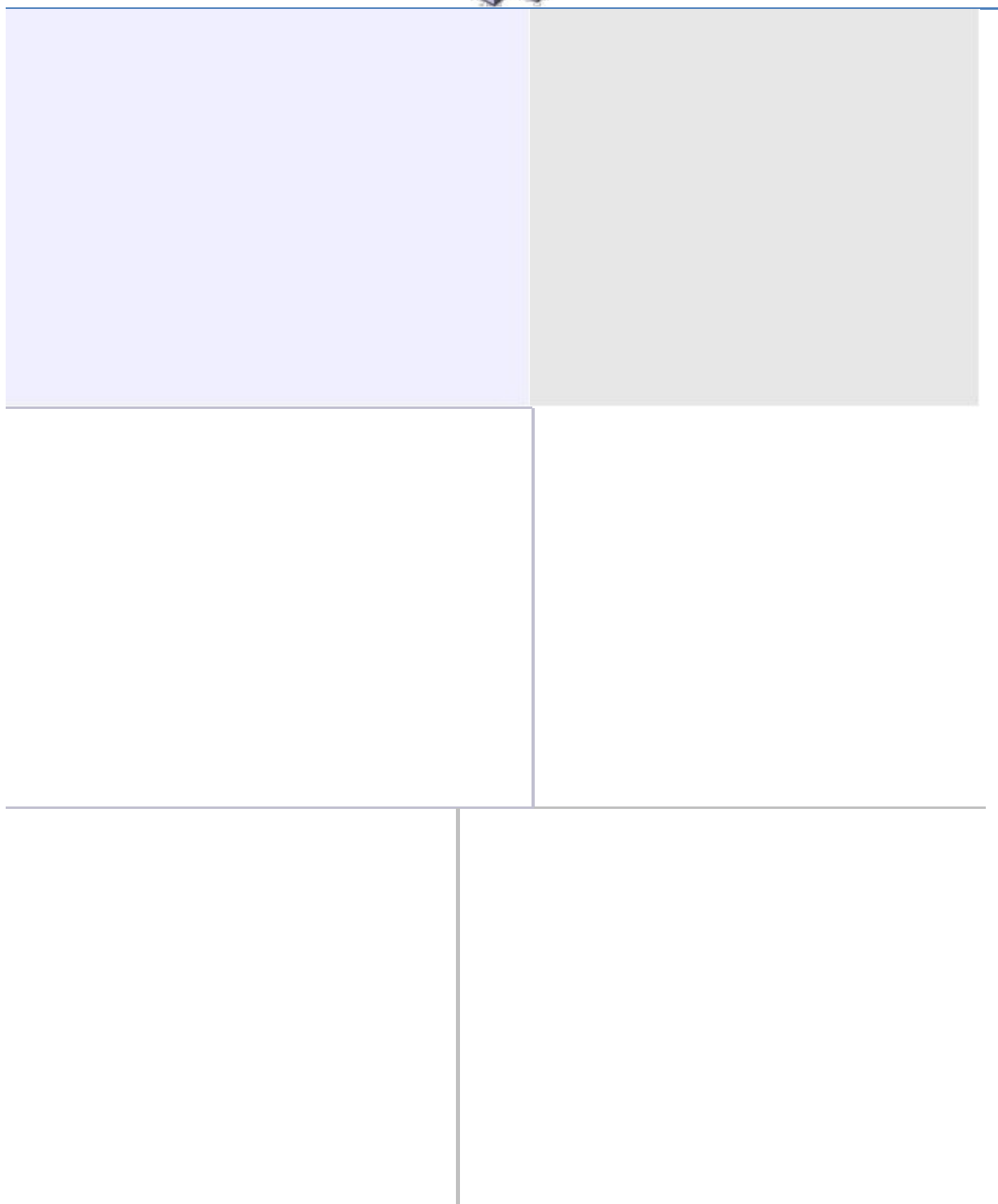
**c has type char and a value of 'z'

void pointers

The void type of pointer is a special type of pointer. In C++, void represents the absence of type, so void pointers are pointers that point to a value that has no type (and thus also an undetermined length and undetermined dereference properties).

This allows void pointers to point to any data type, from an integer value or a float to a string of characters. But in exchange they have a great limitation: the data pointed by them cannot be directly dereferenced (which is logical, since we have no type to dereference to), and for that reason we will always have to cast the address in the void pointer to some other pointer type that points to a concrete data type before dereferencing it.

One of its uses may be to pass generic parameters to a function:



// increaser	y, 1603
#include <iostream>	
using namespace std;	
void increase (void* data, int psize)	
{	
if (psize == sizeof(char))	

```

        { char* pchar; pchar=(char*)data; ++(*pchar); } else if (psize == sizeof(int) )
        { int* pint; pint=(int*)data; ++(*pint); }
    }

int main ()
{
    char a = 'x';

    int b = 1602;
    increase (&a,sizeof(a));

    increase (&b,sizeof(b));
    cout << a << ", " << b << endl;

    return 0;
}

```

sizeof is an operator integrated in the C++ language that returns the size in bytes of its parameter. For non-dynamic data types this value is a constant. Therefore, for example, sizeof(char) is 1, because char type is one byte long.

Null pointer

A null pointer is a regular pointer of any pointer type which has a special value that indicates that it is not pointing to any valid reference or memory address. This value is the result of type-casting the integer value zero to any pointer type.


```

int * p;

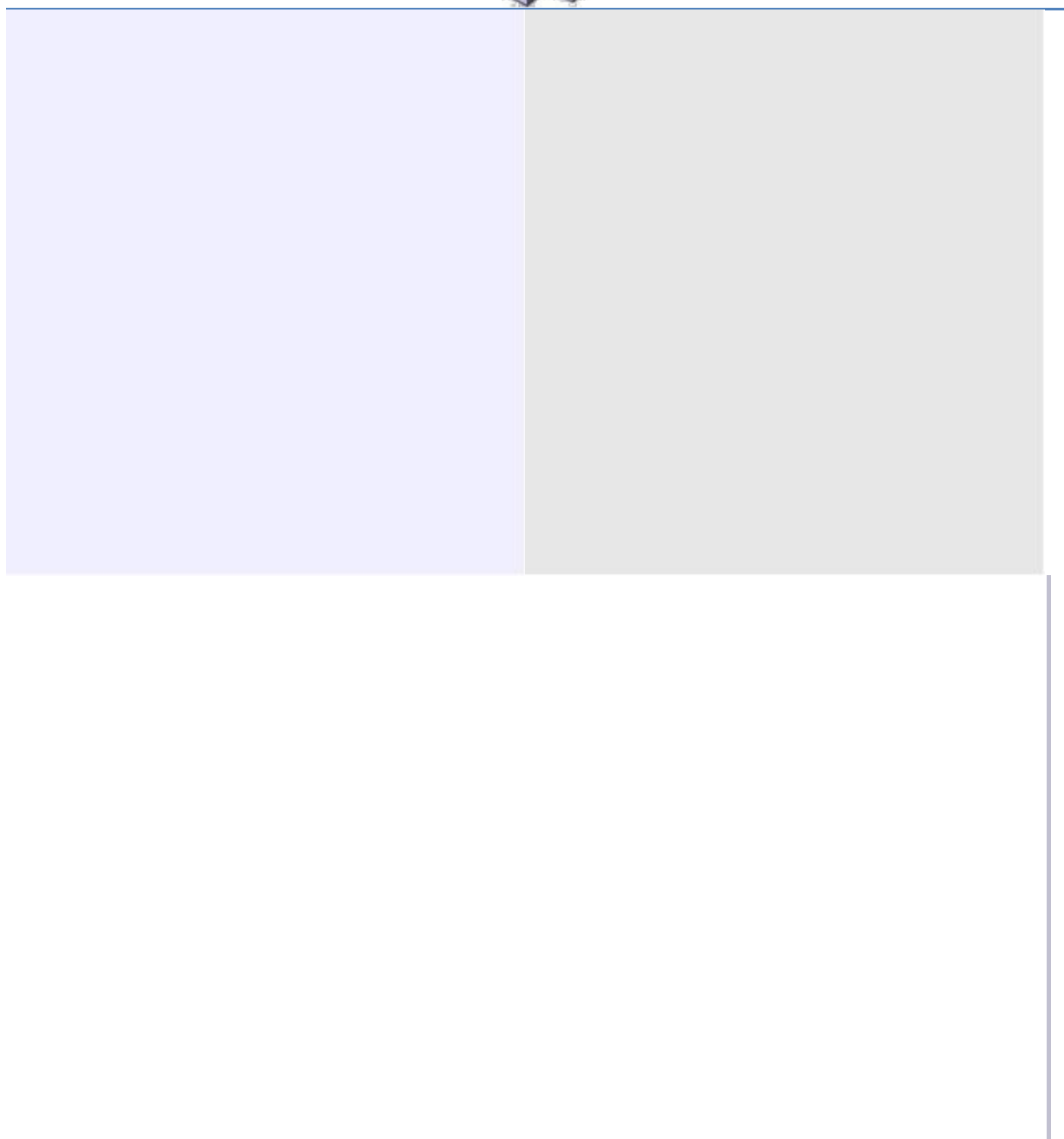
p = 0;    // p has a null pointer value

```

Do not confuse null pointers with void pointers. A null pointer is a value that any pointer may take to represent that it is pointing to "nowhere", while a void pointer is a special type of pointer that can point to somewhere without a specific type. One refers to the value stored in the pointer itself and the other to the type of data it points to.

Pointers to functions

C++ allows operations with pointers to functions. The typical use of this is for passing a function as an argument to another function, since these cannot be passed dereferenced. In order to declare a pointer to a function we have to declare it like the prototype of the function except that the name of the function is enclosed between parentheses () and an asterisk (*) is inserted before the name:



// pointer to functions	8
#include <iostream>	
using namespace std;	

```
int addition (int a, int b)
{ return (a+b); }
```

```
int subtraction (int a, int b)
{ return (a-b); }
```

```
int operation (int x, int y, int
```

```
(*functocall)(int,int))
{
```

```
    int g;
    g = (*functocall)(x,y);
```

```
    return (g);
```

```
}
```

```
int main ()
```

```
{
```

```
    int m,n;
```

```
    int (*minus)(int,int) = subtraction;

    m = operation (7, 5, addition);
    n = operation (20, m, minus);

    cout <<n;
    return 0;

}
```

In the example, minus is a pointer to a function that has two parameters of type int. It is immediately assigned to point to the function subtraction, all in a single line:

```
int (* minus)(int,int) = subtraction;
```



Dynamic Memory

Until now, in all our programs, we have only had as much memory available as we declared for our variables, having the size of all of them to be determined in the source code, before the execution of the program. But, what if we need a variable amount of memory that can only be determined during runtime? For example, in the case that we need some user input to determine the necessary amount of memory space.

The answer is *dynamic memory*, for which C++ integrates the operators `new` and `delete`.

Operators `new` and `new[]`

In order to request dynamic memory we use the operator `new`. `new` is followed by a data type specifier and - if a sequence of more than one element is required- the number of these within brackets `[]`. It returns a pointer to the beginning of the new block of memory allocated. Its form is:

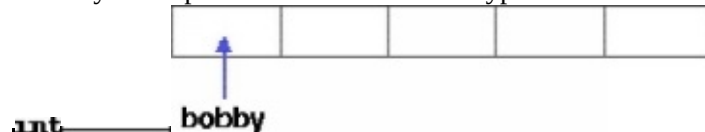
```
pointer = new type  
pointer = new type [number_of_elements]
```

The first expression is used to allocate memory to contain one single element of type `type`. The second one is used to assign a block (an array) of elements of type `type`, where `number_of_elements` is an integer value representing the amount of these. For example:



```
int * bobby;  
bobby = new int [5];
```

In this case, the system dynamically assigns space for five elements of type `int` and returns a pointer to the first element of the sequence, which is assigned to `bobby`. Therefore, now, `bobby` points to a valid block of memory with space for five elements of type `int`.



The first element pointed by bobby can be accessed either with the expression `bobby[0]` or the expression `*bobby`. Both are equivalent as has been explained in the section about pointers. The second element can be accessed either with `bobby[1]` or `*(bobby+1)` and so on...

You could be wondering the difference between declaring a normal array and assigning dynamic memory to a pointer, as we have just done. The most important difference is that the size of an array has to be a constant value, which limits its size to what we decide at the moment of designing the program, before its execution, whereas the dynamic memory allocation allows us to assign memory during the execution of the program (runtime) using any variable or constant value as its size.

The dynamic memory requested by our program is allocated by the system from the memory heap. However, computer memory is a limited resource, and it can be exhausted. Therefore, it is important to have some mechanism to check if our request to allocate memory was successful or not.

C++ provides two standard methods to check if the allocation was successful:

One is by handling exceptions. Using this method an exception of type `bad_alloc` is thrown when the allocation fails. Exceptions are a powerful C++ feature explained later in these tutorials. But for now you should know that if this exception is thrown and it is not handled by a specific handler, the program execution is terminated.



This exception method is the default method used by new, and is the one used in a declaration like:

```
bobby = new int [5];    // if it fails an exception is thrown
```

The other method is known as nothrow, and what happens when it is used is that when a memory allocation fails, instead of throwing a bad_alloc exception or terminating the program, the pointer returned by new is a null pointer, and the program continues its execution.

This method can be specified by using a special object called nothrow, declared in header <new>, as argument for new:

```
bobby = new (nothrow) int [5];
```

In this case, if the allocation of this block of memory failed, the failure could be detected by checking if bobby took a null pointer value:

```
int * bobby;
```

```
bobby = new (nothrow) int [5];
```

```
if (bobby == 0) {
```

```
    error assigning memory. Take measures.
```

```
};
```

This nothrow method requires more work than the exception method, since the value returned has to be checked after each and every memory allocation, but I will use it in our examples due to its simplicity. Anyway this method can become tedious for larger projects, where the exception method is generally preferred. The exception method will be explained in detail later in this tutorial.

Operators delete and delete[]

Since the necessity of dynamic memory is usually limited to specific moments within a program, once it is

no longer needed it should be freed so that the memory becomes available again for other requests of dynamic memory. This is the purpose of the operator delete, whose format is:



`delete` pointer;

`delete []` pointer;

The first expression should be used to delete memory allocated for a single element, and the second one for memory allocated for arrays of elements.

The value passed as argument to delete must be either a pointer to a memory block previously allocated with new, or a null pointer (in the case of a null pointer, delete produces no effect).



// rememb-o-matic	How many numbers would you like to type? 5
#include <iostream>	Enter number : 75
#include <new>	Enter number : 436
using namespace std;	Enter number : 1067
	Enter number : 8
int main ()	Enter number : 32
{	You have entered: 75, 436, 1067, 8, 32,
int i,n;	
int * p;	
cout << "How many numbers would you like to type?" ";	
cin >> i;	
p= new (nothrow) int[i];	
if (p == 0)	
cout << "Error: memory could not be allocated";	
else	
{	
for (n=0; n<i; n++)	
{	
cout << "Enter number: ";	
cin >> p[n];	
}	
cout << "You have entered: ";	
for (n=0; n<i; n++)	
cout << p[n] << ", ";	
delete[] p;	
}	
return 0;	
}	

Notice how the value within brackets in the new statement is a variable value entered by the user (i), not a constant value:

```
p= new (nothrow) int[i];
```

But the user could have entered a value for *i* so big that our system could not handle it. For example, when I tried to give a value of 1 billion to the "How many numbers" question, my system could not allocate that much memory for the program and I got the text message we prepared for this case (Error: memory could not be allocated). Remember that in the case that we tried to allocate the memory without specifying the `nothrow` parameter in the `new` expression, an exception would be thrown, which if it's not handled terminates the program.

It is a good practice to always check if a dynamic memory block was successfully allocated. Therefore, if you use the `nothrow` method, you should always check the value of the pointer returned. Otherwise, use the exception method, even if you do not handle the exception. This way, the program will terminate at that point without causing the unexpected results of continuing executing a code that assumes a block of memory to have been allocated when in fact it has not.

Dynamic memory in ANSI-C

Operators `new` and `delete` are exclusive of C++. They are not available in the C language. But using pure C language and its library, dynamic memory can also be used through the functions `malloc`, `calloc`, `realloc` and `free`, which are also available in C++ including the `<cstdlib>` header file (see `cstdlib` for more info).

The memory blocks allocated by these functions are not necessarily compatible with those returned by `new`, so each one should be manipulated with its own set of functions or operators.



Data structures

We have already learned how groups of sequential data can be used in C++. But this is somewhat restrictive, since in many occasions what we want to store are not mere sequences of elements all of the same data type, but sets of different elements with different data types.

Data structures

A data structure is a group of data elements grouped together under one name. These data elements, known as *members*, can have different types and different lengths. Data structures are declared in C++ using the following syntax:

```
struct structure_name {  
  
    member_type1 member_name1;  
    member_type2 member_name2;  
    member_type3 member_name3;  
    .  
    .  
} object_names;
```

where `structure_name` is a name for the structure type, `object_name` can be a set of valid identifiers for objects that have the type of this structure. Within braces `{ }` there is a list with the data members, each one is specified with a type and a valid identifier as its name.

The first thing we have to know is that a data structure creates a new type: Once a data structure is declared, a new type with the identifier specified as `structure_name` is created and can be used in the rest of the program as if it was any other type. For example:



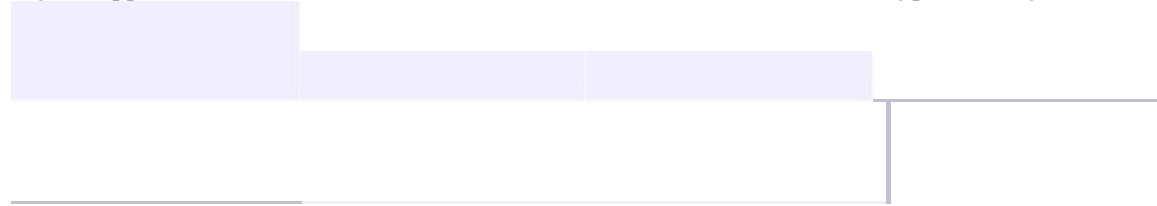
```
struct product {  
    int weight;  
  
    float price;
```

```
};  
  
product apple;  
  
product banana, melon;
```

We have first declared a structure type called `product` with two members: `weight` and `price`, each of a different fundamental type. We have then used this name of the structure type (`product`) to declare three objects of that type: `apple`, `banana` and `melon` as we would have done with any fundamental data type.

Once declared, `product` has become a new valid type name like the fundamental ones `int`, `char` or `short` and from that point on we are able to declare objects (variables) of this compound new type, like we have done with `apple`, `banana` and `melon`.

Right at the end of the struct declaration, and before the ending semicolon, we can use the optional field `object_name` to directly declare objects of the structure type. For example, we can also declare the structure objects `apple`, `banana` and `melon` at the moment we define the data structure type this way:



```
struct product {  
    int weight;  
    float price;  
} apple, banana, melon;
```

It is important to clearly differentiate between what is the structure type name, and what is an object (variable) that has this structure type. We can instantiate many objects (i.e. variables, like `apple`, `banana` and `melon`) from a single structure type (`product`).



Once we have declared our three objects of a determined structure type (apple, banana and melon) we can operate directly with their members. To do that we use a dot (.) inserted between the object name and the member name. For example, we could operate with any of these elements as if they were standard variables of their respective types:

```
apple.weight  
apple.price
```

```
banana.weight  
banana.price
```

```
melon.weight
```

```
melon.price
```

Each one of these has the data type corresponding to the member they refer to: apple.weight, banana.weight and melon.weight are of type int, while apple.price, banana.price and melon.price are of type float.

Let's see a real example where you can see how a structure type can be used in the same way as fundamental types:

// example about structures	Enter title: Alien
#include <iostream>	Enter year: 1979
#include <string>	
#include <sstream>	My favorite movie is:
using namespace std;	2001 A Space Odyssey (1968)
	And yours is:
struct movies_t {	Alien (1979)
string title;	
int year;	
} mine, yours;	
void printmovie (movies_t movie);	
int main ()	
{	

string mystr;	
mine.title = "2001 A Space Odyssey";	
mine.year = 1968;	
cout << "Enter title: ";	
getline (cin,yours.title);	
cout << "Enter year: ";	
getline (cin,mystr);	
stringstream(mystr) >> yours.year;	
cout << "My favorite movie is:\n ";	
printmovie (mine);	
cout << "And yours is:\n ";	
printmovie (yours);	
return 0;	
}	
void printmovie (movies_t movie)	
{	
cout << movie.title;	
cout << " (" << movie.year << ")\n";	
}	

The example shows how we can use the members of an object as regular variables. For example, the member yours.year is a valid variable of type int, and mine.title is a valid variable of type string.

The objects mine and yours can also be treated as valid variables of type movies_t, for example we have passed them to the function printmovie as we would have done with regular variables. Therefore, one of the most important advantages of data structures is that we can either refer to their members individually or to the entire structure as a block with only one identifier.



Data structures are a feature that can be used to represent databases, especially if we consider the possibility of building arrays of them:



// array	of structures	Enter title: Blade Runner
#include	<iostream>	Enter year: 1982
#include	<string>	Enter title: Matrix
#include <sstream>		Enter year: 1999
using namespace std;		Enter title: Taxi Driver
		Enter year: 1976
#define N_MOVIES 3		
		You have entered these movies:
struct movies_t {		Blade Runner (1982)
string title;		Matrix (1999)
int year;		Taxi Driver (1976)
} films [N_MOVIES];		
void printmovie (movies_t movie);		
int main ()		
{		
string mystr;		

```

int n;

for (n=0; n<N_MOVIES; n++)
{
    cout << "Enter title: ";

    getline (cin,films[n].title);
    cout << "Enter year: ";

    getline (cin,mystr);
    stringstream(mystr) >> films[n].year;

}

cout << "\nYou have entered these movies:\n";
for (n=0; n<N_MOVIES; n++)

    printmovie (films[n]);

return 0;

}

void printmovie (movies_t movie)
{

    cout << movie.title;
    cout << " (" << movie.year << ")\n";

}

```

Pointers to structures

Like any other type, structures can be pointed by its own type of pointers:



```

struct movies_t {

    string title;
    int year;

};

```

```
movies_t amovie;  
movies_t * pmovie;
```

Here amovie is an object of structure type movies_t, and pmovie is a pointer to point to objects of structure type movies_t. So, the following code would also be valid:

```
pmovie = &amovie;
```

The value of the pointer pmovie would be assigned to a reference to the object amovie (its memory address).



We will now go with another example that includes pointers, which will serve to introduce a new operator: the arrow operator (\rightarrow):



```
pointers to structures #include <iostream> #include <string> #include <sstream> using namespace std;
```

```
    string title;
```

```
    int year;
```

```
};

int main ()
{
    string mystr;

    movies_t amovie;

    movies_t * pmovie;
    pmovie = &amovie;

    cout << "Enter title: ";

    getline (cin, pmovie->title);

    cout << "Enter year: ";
    getline (cin, mystr);

    (stringstream) mystr >> pmovie->year;

    cout << "\nYou have entered:\n"; cout << pmovie->title;

    cout << " (" << pmovie->year << ")\n";

    return 0;
}
```

Enter title: Invasion of the body snatchers

Enter year: 1978

You have entered:

Invasion of the body snatchers (1978)

The previous code includes an important introduction: the arrow operator (->). This is a dereference operator that is used exclusively with pointers to objects with members. This operator serves to access a member of an object to which we have a reference. In the example we used:

pmovie->title

Which is for all purposes equivalent to:

(*pmovie).title

Both expressions pmovie->title and (*pmovie).title are valid and both mean that we are evaluating the member title of the data structure pointed by a pointer called pmovie. It must be clearly differentiated from:

`*pmovie.title`

which is equivalent to:

`*(pmovie.title)`

And that would access the value pointed to by a hypothetical pointer member called `title` of the structure object `pmovie` (which in this case would not be a pointer). The following panel summarizes possible combinations of pointers and structure members:



Expression	What is evaluated	Equivalent
a.b	Member b of object a	
a->b	Member b of object pointed by a	(*a).b
a.b	Value pointed by member b of object	a(a.b)

Nesting structures

Structures can also be nested so that a valid element of a structure can also be in its turn another structure.



```
struct movies_t {
```

```
    string title;
```

```
    int year;
```

```
};
```

```
struct friends_t {
```

```
    string name;
```



```
string email;
```

```
movies_t favorite_movie;
```

```
} charlie, maria;
```

```
friends_t * pfriends = &charlie;
```

After the previous declaration we could use any of the following expressions:

```
charlie.name
```

```
maria.favorite_movie.title
```

```
charlie.favorite_movie.year
```

```
pfriends->favorite_movie.year
```

(where, by the way, the last two expressions refer to the same member).



Other Data Types

Defined data types (typedef)

C++ allows the definition of our own types based on other existing data types. We can do this using the keyword typedef, whose format is:

```
typedef existing_type new_type_name ;
```

where existing_type is a C++ fundamental or compound type and new_type_name is the name for the new type we are defining. For example:

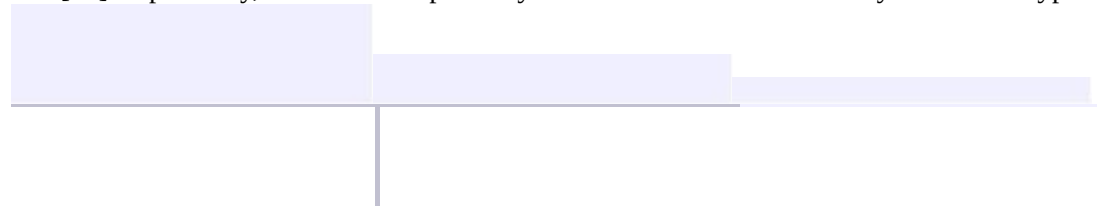


```
typedef char C;
```

```
typedef unsigned int WORD;  
typedef char * pChar;
```

```
typedef char field [50];
```

In this case we have defined four data types: C, WORD, pChar and field as char, unsigned int, char* and char[50] respectively, that we could perfectly use in declarations later as any other valid type:



```
C mychar, anotherchar, *ptc1;  
WORD myword;
```

```
pChar ptc2;  
field name;
```

typedef does not create different types. It only creates synonyms of existing types. That means that the type of myword can be considered to be either WORD or unsigned int, since both are in fact the same type.

typedef can be useful to define an alias for a type that is frequently used within a program. It is also useful to define types when it is possible that we will need to change the type in later versions of our program, or if a type you want to use has a name that is too long or confusing.

Unions

Unions allow one same portion of memory to be accessed as different data types, since all of them are in fact the same location in memory. Its declaration and use is similar to the one of structures but its functionality is totally different:

```
union union_name {  
    member_type1 member_name1;  
  
    member_type2 member_name2;  
    member_type3 member_name3;  
  
    .  
    .  
}  
object_names;
```

All the elements of the union declaration occupy the same physical space in memory. Its size is the one of the greatest element of the declaration. For example:



```
union mytypes_t {  
    char c;  
    int i;  
    float f;  
} mytypes;
```

defines three elements:



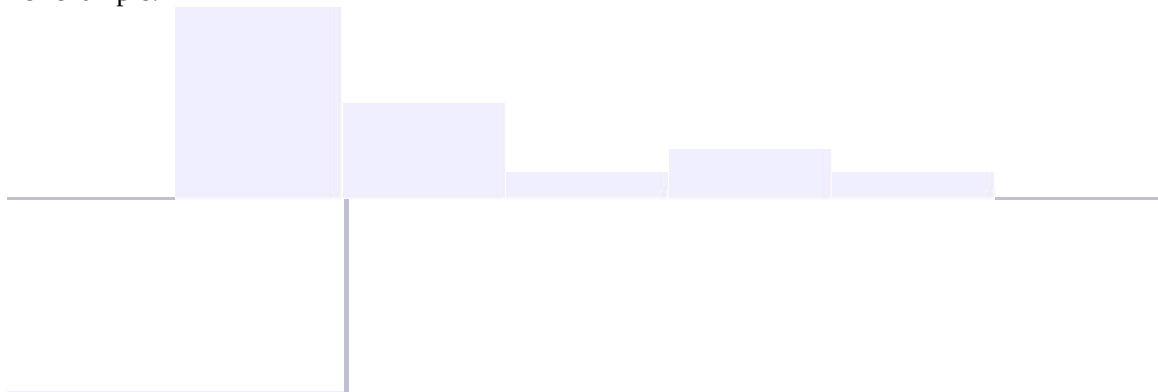
mytypes.c
mytypes.i

mytypes.f

each one with a different data type. Since all of them are referring to the same location in memory, the modification of one of the elements will affect the value of all of them. We cannot store different values in them independent of each other.

One of the uses a union may have is to unite an elementary type with an array or structures of smaller elements.

For example:



```

union mix_t {

    long l;
    struct {

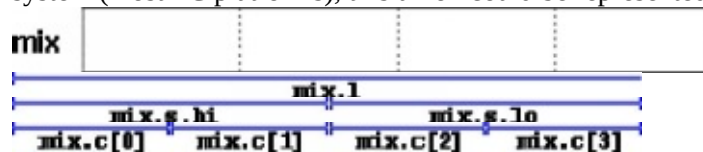
        short hi;
        short lo;

    } s; char c[4];

} mix;

```

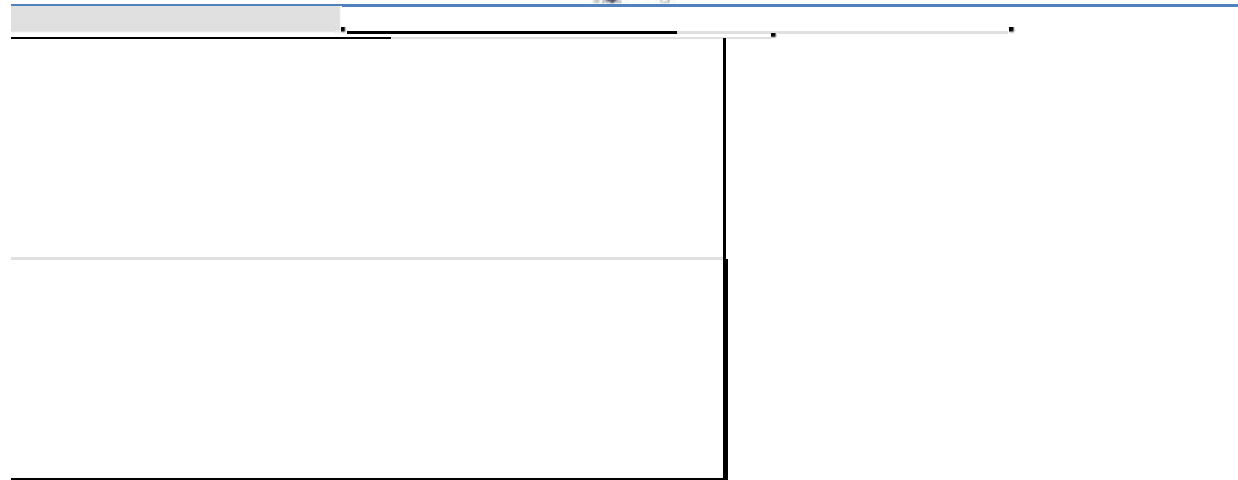
defines three names that allow us to access the same group of 4 bytes: `mix.l`, `mix.s` and `mix.c` and which we can use according to how we want to access these bytes, as if they were a single long-type data, as if they were two short elements or as an array of char elements, respectively. I have mixed types, arrays and structures in the union so that you can see the different ways that we can access the data. For a *little-endian* system (most PC platforms), this union could be represented as:



The exact alignment and order of the members of a union in memory is platform dependant. Therefore be aware of possible portability issues with this type of use.

Anonymous unions

In C++ we have the option to declare anonymous unions. If we declare a union without any name, the union will be anonymous and we will be able to access its members directly by their member names. For example, look at the difference between these two structure declarations:



structure with regular union structure with anonymous union

struct {	struct {
char title[50];	char title[50];
char author[50];	char author[50];
union {	union {
float dollars;	float dollars;
int yens;	int yens;
} price;	};
} book;	} book;

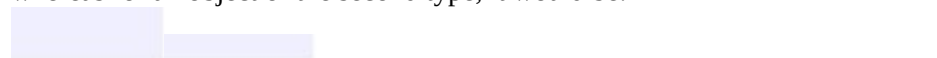
The only difference between the two pieces of code is that in the first one we have given a name to the union (price) and in the second one we have not. The difference is seen when we access the members dollars and yens of an object of this type. For an object of the first type, it would be:



book.price.dollars

book.price.yens

whereas for an object of the second type, it would be:



book.dollars

book.yens

Once again I remind you that because it is a union and not a struct, the members dollars and yens occupy the same physical space in the memory so they cannot be used to store two different values simultaneously. You can set a value for price in dollars or in yens, but not in both.

Enumerations (enum)

Enumerations create new data types to contain something different that is not limited to the values fundamental data types may take. Its form is the following:

```
enum enumeration_name {  
    value1,  
  
    value2,  
    value3,  
  
    .  
    .  
  
} object_names;
```

For example, we could create a new type of variable called color to store colors with the following declaration:

```
enum colors_t {black, blue, green, cyan, red, purple, yellow, white};
```

Notice that we do not include any fundamental data type in the declaration. To say it somehow, we have created a whole new data type from scratch without basing it on any other existing type. The possible values that variables of this new type color_t may take are the new constant values included within braces. For example, once the colors_t enumeration is declared the following expressions will be valid:

```
colors_t mycolor;
```

```
mycolor = blue;  
if (mycolor == green) mycolor = red;
```

Enumerations are type compatible with numeric variables, so their constants are always assigned an integer numerical value internally. If it is not specified, the integer value equivalent to the first possible value is equivalent to 0 and the following ones follow a +1 progression. Thus, in our data type colors_t that we have defined above, black would be equivalent to 0, blue would be equivalent to 1, green to 2, and so on.



We can explicitly specify an integer value for any of the constant values that our enumerated type can take. If the constant value that follows it is not given an integer value, it is automatically assumed the same value as the previous one plus one. For example:



```
enum months_t { january=1, february, march, april, may, june, july, august,  
                september,  october,  november,  
                december} y2k;
```

In this case, variable y2k of enumerated type months_t can contain any of the 12 possible values that go from january to december and that are equivalent to values between 1 and 12 (not between 0 and 11, since we have made january equal to 1).



Object Oriented Programming

Classes (I)

A *class* is an expanded concept of a data structure: instead of holding only data, it can hold both data and functions.

An *object* is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.

Classes are generally declared using the keyword `class`, with the following format:

```
class class_name {  
    access_specifier_1:  
        member1;  
    access_specifier_2:  
        member2;  
    ...  
} object_names;
```

Where `class_name` is a valid identifier for the class, `object_names` is an optional list of names for objects of this class. The body of the declaration can contain members, that can be either data or function declarations, and optionally access specifiers.

All is very similar to the declaration on data structures, except that we can now include also functions and members, but also this new thing called *access specifier*. An access specifier is one of the following three keywords: `private`, `public` or `protected`. These specifiers modify the access rights that the members following them acquire:

private members of a class are accessible only from within other members of the same class or from their *friends*.

protected members are accessible from members of their same class and from their friends, but also from members of their derived classes.

Finally, public members are accessible from anywhere where the object is visible.

By default, all members of a class declared with the `class` keyword have private access for all its members.

Therefore, any member that is declared before one other class specifier automatically has private access. For example:



```
class CRectangle {  
    int x, y;  
    public:  
        void set_values (int,int);  
        int area (void);  
} rect;
```

Declares a class (i.e., a type) called CRectangle and an object (i.e., a variable) of this class called rect. This class contains four members: two data members of type int (member x and member y) with private access (because private is the default access level) and two member functions with public access: set_values() and area(), of which for now we have only included their declaration, not their definition.

Notice the difference between the class name and the object name: In the previous example, CRectangle was the class name (i.e., the type), whereas rect was an object of type CRectangle. It is the same relationship int and a have in the following declaration:



```
int a;
```

where int is the type name (the class) and a is the variable name (the object).

After the previous declarations of CRectangle and rect, we can refer within the body of the program to any of the public members of the object rect as if they were normal functions or normal variables, just by putting the object's name followed by a dot (.) and then the name of the member. All very similar to what we did with plain data structures before. For example:

```
rect.set_values (3,4);  
myarea = rect.area();
```

The only members of rect that we cannot access from the body of our program outside the class are x and y, since they have private access and they can only be referred from within other members of that same class.

Here is the complete example of class CRectangle:	
// classes example	area: 12
#include <iostream>	
using namespace std;	
class CRectangle {	
int x, y;	
public:	
void set_values (int,int);	
int area () {return (x*y);}	
};	
void CRectangle::set_values (int a, int b) {	
x = a;	
y = b;	
}	
int main () {	
CRectangle rect;	

rect.set_values (3,4);	
cout << "area: " << rect.area();	
return 0;	
}	

The most important new thing in this code is the operator of scope (::, two colons) included in the definition of set_values(). It is used to define a member of a class from outside the class definition itself.

You may notice that the definition of the member function area() has been included directly within the definition of the CRectangle class given its extreme simplicity, whereas set_values() has only its prototype declared within the class, but its definition is outside it. In this outside declaration, we must use the operator of scope (::) to specify that we are defining a function that is a member of the class CRectangle and not a regular global function.

The scope operator (::) specifies the class to which the member being declared belongs, granting exactly the same scope properties as if this function definition was directly included within the class definition. For example, in the function set_values() of the previous code, we have been able to use the variables x and y, which are private members of class CRectangle, which means they are only accessible from other members of their class.

The only difference between defining a class member function completely within its class or to include only the prototype and later its definition, is that in the first case the function will automatically be considered an inline member function by the compiler, while in the second it will be a normal (not-inline) class member function, which in fact supposes no difference in behavior.

Members x and y have private access (remember that if nothing else is said, all members of a class defined with keyword class have private access). By declaring them private we deny access to them from anywhere outside the



class. This makes sense, since we have already defined a member function to set values for those members within the object: the member function `set_values()`. Therefore, the rest of the program does not need to have direct access to them. Perhaps in a so simple example as this, it is difficult to see an utility in protecting those two variables, but in greater projects it may be very important that values cannot be modified in an unexpected way (unexpected from the point of view of the object).

One of the greater advantages of a class is that, as any other type, we can declare several objects of it. For example, following with the previous example of class `CRectangle`, we could have declared the object `rectb` in addition to the object `rect`:

// example: one class, two objects	rect area: 12
#include <iostream>	rectb area: 30
using namespace std;	
class CRectangle {	
int x, y;	
public:	
void set_values (int,int);	
int area () {return (x*y);}	
};	
void CRectangle::set_values (int a, int b) {	
x = a;	
y = b;	
}	
int main () {	
CRectangle rect, rectb;	
rect.set_values (3,4);	
rectb.set_values (5,6);	
cout << "rect area: " << rect.area() << endl;	
cout << "rectb area: " << rectb.area() << endl;	
return 0;	
}	

In this concrete case, the class (type of the objects) to which we are talking about is `CRectangle`, of which there are two instances or objects: `rect` and `rectb`. Each one of them has its own member variables and member functions.

Notice that the call to `rect.area()` does not give the same result as the call to `rectb.area()`. This is because each object of class `CRectangle` has its own variables `x` and `y`, as they, in some way, have also their own function members `set_value()` and `area()` that each uses its object's own variables to operate.

That is the basic concept of *object-oriented programming*: Data and functions are both members of the object. We no longer use sets of global variables that we pass from one function to another as parameters, but instead we handle objects that have their own data and functions embedded as members. Notice that we have not had to give any parameters in any of the calls to `rect.area` or `rectb.area`. Those member functions directly used the data members of their respective objects `rect` and `rectb`.

Constructors and destructors

Objects generally need to initialize variables or assign dynamic memory during their process of creation to become operative and to avoid returning unexpected values during their execution. For example, what would happen if in the previous example we called the member function `area()` before having called function `set_values()`? Probably we would have gotten an undetermined result since the members `x` and `y` would have never been assigned a value.

In order to avoid that, a class can include a special function called constructor, which is automatically called whenever a new object of this class is created. This constructor function must have the same name as the class, and cannot have any return type; not even `void`.

We are going to implement `CRectangle` including a constructor:



// example: class constructor	rect area: 12
#include <iostream>	rectb area: 30
using namespace std;	
class CRectangle {	
int width, height;	
public:	
CRectangle (int,int);	
int area () {return (width*height);}	
};	
CRectangle::CRectangle (int a, int b) {	
width = a;	
height = b;	
}	
int main () {	
CRectangle rect (3,4);	
CRectangle rectb (5,6);	
cout << "rect area: " << rect.area() << endl;	
cout << "rectb area: " << rectb.area() << endl;	
return 0;	
}	

As you can see, the result of this example is identical to the previous one. But now we have removed the member function `set_values()`, and have included instead a constructor that performs a similar action: it initializes the values of `x` and `y` with the parameters that are passed to it.

Notice how these arguments are passed to the constructor at the moment at which the objects of this class are created:

```
CRectangle rect (3,4);
```

```
CRectangle rectb (5,6);
```

Constructors cannot be called explicitly as if they were regular member functions. They are only executed

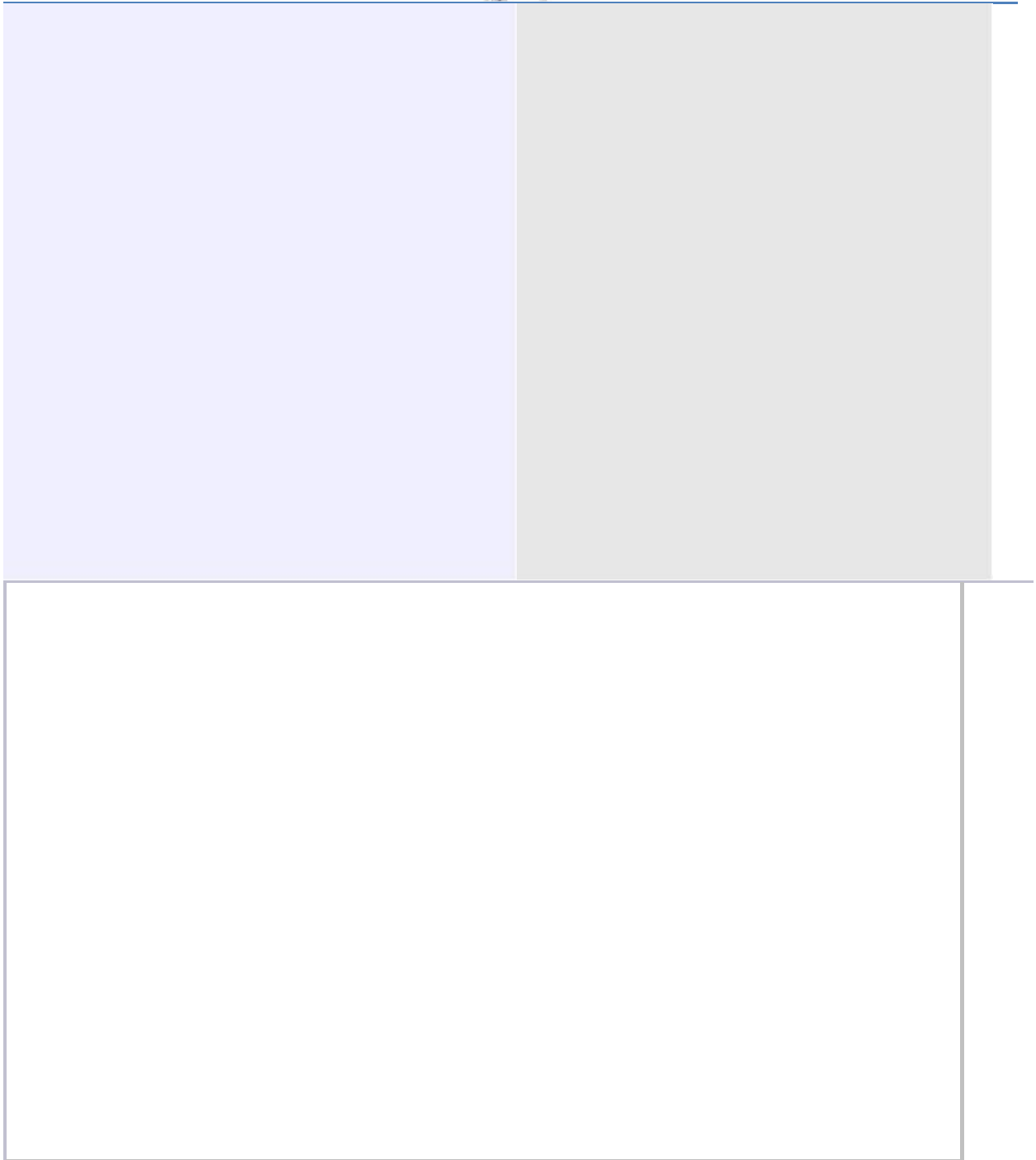
when a new object of that class is created.

You can also see how neither the constructor prototype declaration (within the class) nor the latter constructor definition include a return value; not even void.

The *destructor* fulfills the opposite functionality. It is automatically called when an object is destroyed, either because its scope of existence has finished (for example, if it was defined as a local object within a function and the function ends) or because it is an object dynamically assigned and it is released using the operator delete.

The destructor must have the same name as the class, but preceded with a tilde sign (~) and it must also return no value.

The use of destructors is especially suitable when an object assigns dynamic memory during its lifetime and at the moment of being destroyed we want to release the memory that the object was allocated.



// example on constructors and destructors

rect area: 12

```

#include <iostream>                                rectb area: 30

using namespace std;

class CRectangle {
    int *width, *height;

    public:

    CRectangle (int,int);
    ~CRectangle ();

    int area () {return (*width * *height);}
};

CRectangle::CRectangle (int a, int b) {

    width = new int;
    height = new int;

    *width = a;
    *height = b;

}

CRectangle::~~CRectangle () {
    delete width;

    delete height;
}

int main () {

    CRectangle rect (3,4), rectb (5,6);
    cout << "rect area: " << rect.area() << endl;

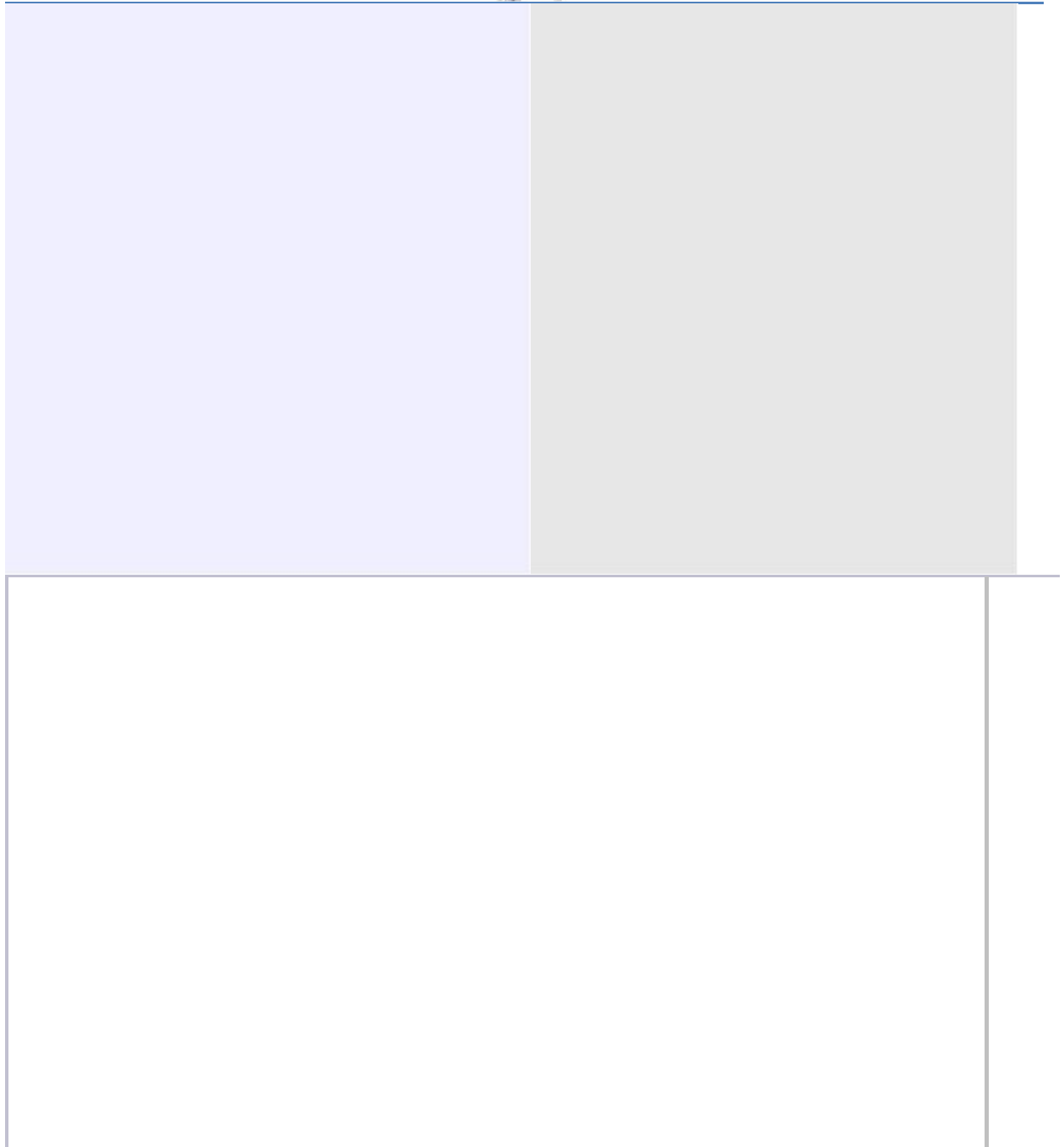
    cout << "rectb area: " << rectb.area() << endl;
    return 0;

}

```

Overloading Constructors

Like any other function, a constructor can also be overloaded with more than one function that have the same name but different types or number of parameters. Remember that for overloaded functions the compiler will call the one whose parameters match the arguments used in the function call. In the case of constructors, which are automatically called when an object is created, the one executed is the one that matches the arguments passed on the object declaration:



```
// overloading class constructors  
#include <iostream>
```

```
rect area: 12  
rectb area: 25
```

```

using namespace std;

class CRectangle {
    int width, height;

public:
    CRectangle ();
    CRectangle (int,int);

    int area (void) {return (width*height);}
};

CRectangle::CRectangle () {
    width = 5;
    height = 5;
}

CRectangle::CRectangle (int a, int b) {
    width = a;

    height = b;
}

int main () {

    CRectangle rect (3,4);
    CRectangle rectb;

    cout << "rect area: " << rect.area() << endl;
    cout << "rectb area: " << rectb.area() << endl;

    return 0;
}

```

In this case, rectb was declared without any arguments, so it has been initialized with the constructor that has no parameters, which initializes both width and height with a value of 5.

Important: Notice how if we declare a new object and we want to use its default constructor (the one without parameters), we do not include parentheses ():

CRectangle	rectb;	//	right
CRectangle	rectb();	//	wrong!

Default constructor

If you do not declare any constructors in a class definition, the compiler assumes the class to have a default constructor with no arguments. Therefore, after declaring a class like this one:


```
class CExample {  
    public:  
  
        int a,b,c;  
  
        void multiply (int n, int m) { a=n; b=m; c=a*b; }; };
```

The compiler assumes that CExample has a default constructor, so you can declare objects of this class by simply declaring them without any arguments:

```
CExample ex;
```

But as soon as you declare your own constructor for a class, the compiler no longer provides an implicit default constructor. So you have to declare all objects of that class according to the constructor prototypes you defined for the class:



```
class CExample {  
    public:  
        int a,b,c;  
        CExample (int n, int m) { a=n; b=m; };  
        void multiply () { c=a*b; };  
};
```

Here we have declared a constructor that takes two parameters of type int. Therefore the following object declaration would be correct:

```
CExample ex (2,3);
```

But,

```
CExample ex;
```

Would not be correct, since we have declared the class to have an explicit constructor, thus replacing the default constructor.

But the compiler not only creates a default constructor for you if you do not specify your own. It provides three special member functions in total that are implicitly declared if you do not declare your own. These are the *copy constructor*, the *copy assignment operator*, and the default destructor.

The copy constructor and the copy assignment operator copy all the data contained in another object to the data members of the current object. For CExample, the copy constructor implicitly declared by the compiler would be something similar to:

```
CExample::CExample (const CExample& rv) {
```

```
    a=rv.a;  b=rv.b;  c=rv.c;
```

```
}
```

Therefore, the two following object declarations would be correct:

CExample	ex (2,3);	
CExample	ex2 (ex);	// copy constructor (data copied from ex)

Pointers to classes

It is perfectly valid to create pointers that point to classes. We simply have to consider that once declared, a class becomes a valid type, so we can use the class name as the type for the pointer. For example:

```
CRectangle * prect;
```

is a pointer to an object of class CRectangle.

As it happened with data structures, in order to refer directly to a member of an object pointed by a pointer we can use the arrow operator (->) of indirection. Here is an example with some possible combinations:



// pointer to classes example	a area: 2
#include <iostream>	*b area: 12
using namespace std;	*c area: 2
	d[0] area: 30
class CRectangle {	d[1] area: 56
int width, height;	
public:	
void set_values (int, int);	
int area (void) {return (width * height);}	
};	
void CRectangle::set_values (int a, int b) {	
width = a;	
height = b;	
}	
int main () {	
CRectangle a, *b, *c;	
CRectangle * d = new CRectangle[2];	
b= new CRectangle;	
c= &a;	
a.set_values (1,2);	
b->set_values (3,4);	
d->set_values (5,6);	
d[1].set_values (7,8);	
cout << "a area: " << a.area() << endl;	
cout << "*b area: " << b->area() << endl;	
cout << "*c area: " << c->area() << endl;	
cout << "d[0] area: " << d[0].area() << endl;	
cout << "d[1] area: " << d[1].area() << endl;	
delete[] d;	
delete b;	
return 0;	
}	

Next you have a summary on how can you read some pointer and class operators (*, &, ., ->, []) that appear in the previous example:

expression	can be read as
*x	pointed by x
&x	address of x
x.y	member y of object x
x->y	member y of object pointed by x
(*x).y	member y of object pointed by x (equivalent to the previous one)
x[0]	first object pointed by x
x[1]	second object pointed by x
x[n]	(n+1)th object pointed by x

Be sure that you understand the logic under all of these expressions before proceeding with the next sections. If you have doubts, read again this section and/or consult the previous sections about pointers and data structures.

Classes defined with struct and union

Classes can be defined not only with keyword class, but also with keywords struct and union.

The concepts of class and data structure are so similar that both keywords (struct and class) can be used in C++ to declare classes (i.e. structs can also have function members in C++, not only data members). The only difference between both is that members of classes declared with the keyword struct have public access by default, while members of classes declared with the keyword class have private access. For all other purposes both keywords are equivalent.



The concept of unions is different from that of classes declared with struct and class, since unions only store one data member at a time, but nevertheless they are also classes and can thus also hold function members. The default access in union classes is public.

Overloadable operators

+	-	*	/	=	<	>	+=	-	*=	/=	
<<=	>>=	==	!=	<=	>=	++	--	%	&	^	
~	&=	^=	=	&&		%=	[]	()	,	-	>*
delete		new[]		delete[]							

To overload an operator in order to use it with classes we declare *operator functions*, which are regular functions whose names are the operator keyword followed by the operator sign that we want to overload. The format is:

```
type operator sign (parameters) { /*...*/ }
```

Here you have an example that overloads the addition operator (+). We are going to create a class to store bidimensional vectors and then we are going to add two of them: a(3,1) and b(1,2). The addition of two bidimensional vectors is an operation as simple as adding the two x coordinates to obtain the resulting x coordinate and adding the two y coordinates to obtain the resulting y. In this case the result will be (3+1,1+2) = (4,3).



// vectors: overloading operators example	4,3
#include <iostream>	
using namespace std;	
class CVector {	
public:	
int x,y;	
CVector () {};	
CVector (int,int);	
CVector operator + (CVector);	
};	
CVector::CVector (int a, int b) {	
x = a;	
y = b;	
}	
CVector CVector::operator+ (CVector param) {	
CVector temp;	
temp.x = x + param.x;	
temp.y = y + param.y;	
return (temp);	
}	
int main () {	
CVector a (3,1);	
CVector b (1,2);	
CVector c;	
c = a + b;	
cout << c.x << " " << c.y;	
return 0;	
}	

It may be a little confusing to see so many times the CVector identifier. But, consider that some of them refer to the class name (type) CVector and some others are functions with that name (constructors must have the same name as the class). Do not confuse them:



CVector	(int, int);	//	function	name CVector (constructor)
CVector	operator+ (CVector);	//	function	returns a CVector

The function `operator+` of class `CVector` is the one that is in charge of overloading the addition operator (+). This function can be called either implicitly using the operator, or explicitly using the function name:



```
c = a + b;
c = a.operator+ (b);
```

Both expressions are equivalent.

Notice also that we have included the empty constructor (without parameters) and we have defined it with an empty block:

```
CVector () { };
```

This is necessary, since we have explicitly declared another constructor:

```
CVector (int, int);
```

And when we explicitly declare any constructor, with any number of parameters, the default constructor with no parameters that the compiler can declare automatically is not declared, so we need to declare it ourselves in order to be able to construct objects of this type without parameters. Otherwise, the declaration:



```
CVector c;
```

included in main() would not have been valid.

Anyway, I have to warn you that an empty block is a bad implementation for a constructor, since it does not fulfill the minimum functionality that is generally expected from a constructor, which is the initialization of all the member variables in its class. In our case this constructor leaves the variables x and y undefined. Therefore, a more advisable definition would have been something similar to this:

```
CVector () { x=0; y=0; };
```

which in order to simplify and show only the point of the code I have not included in the example.

As well as a class includes a default constructor and a copy constructor even if they are not declared, it also includes a default definition for the assignment operator (=) with the class itself as parameter. The behavior which is defined by default is to copy the whole content of the data members of the object passed as argument (the one at the right side of the sign) to the one at the left side:

```
CVector d (2,3);
```

```
CVector e;
```

```
e = d;           // copy assignment operator
```

The copy assignment operator function is the only operator member function implemented by default. Of course, you can redefine it to any other functionality that you want, like for example, copy only certain class members or perform additional initialization procedures.

The overload of operators does not force its operation to bear a relation to the mathematical or usual meaning of the operator, although it is recommended. For example, the code may not be very intuitive if you use operator + to subtract two classes or operator == to fill with zeros a class, although it is perfectly possible to do so.

Although the prototype of a function operator+ can seem obvious since it takes what is at the right side of the operator as the parameter for the operator member function of the object at its left side, other operators may not be so obvious. Here you have a table with a summary on how the different operator functions have to be declared (replace @ by the operator in each case):

Expression	Operator	Member function	Global function
@a	+ - * & ! ~ ++ --	A::operator@()	operator@(A)
a@	++ --	A::operator@(int)	operator@(A,int)
a@b	+ - * / % ^ & < > == != <= >= << >> &&	A::operator@ (B)	operator@(A,B)
a@b	= += -= *= /= %= ^= &= = <<= >>= []	A::operator@ (B)	-
a(b, c...)	()	A::operator() (B, C...)	-
a->x	->	A::operator->()	-

Where a is an object of class A, b is an object of class B and c is an object of class C.

You can see in this panel that there are two ways to overload some class operators: as a member function and as a global function. Its use is indistinct, nevertheless I remind you that functions that are not members of a class cannot access the private or protected members of that class unless the global function is its friend (friendship is explained later).

The keyword this

The keyword this represents a pointer to the object whose member function is being executed. It is a pointer to the object itself.



One of its uses can be to check if a parameter passed to a member function is the object itself. For example,

// this	yes, &a is b
#include <iostream>	
using namespace std;	
class CDummy {	
public:	
int isitme (CDummy& param);	
};	
int CDummy::isitme (CDummy& param)	
{	
if (¶m == this) return true;	
else return false;	
}	
int main () {	
CDummy a;	
CDummy* b = &a;	
if (b->isitme(a))	
cout << "yes, &a is b";	
return 0;	
}	

It is also frequently used in operator= member functions that return objects by reference (avoiding the use of temporary objects). Following with the vector's examples seen before we could have written an operator= function similar to this one:



CVector& CVector::operator= (const CVector& param)

```
{  
    x=param.x;  
    y=param.y;  
    return *this;  
}
```

In fact this function is very similar to the code that the compiler generates implicitly for this class if we do not include an `operator=` member function to copy objects of this class.

Static members

A class can contain *static* members, either data or functions.

Static data members of a class are also known as "class variables", because there is only one unique value for all the objects of that same class. Their content is not different from one object of this class to another.

For example, it may be used for a variable within a class that can contain a counter with the number of objects of that class that are currently allocated, as in the following example:



// static members in classes	7
#include <iostream>	6
using namespace std;	
class CDummy {	
public:	
static int n;	
CDummy () { n++; };	
~CDummy () { n--; };	
};	
int CDummy::n=0;	
int main () {	
CDummy a;	
CDummy b[5];	
CDummy * c = new CDummy;	
cout << a.n << endl;	
delete c;	
cout << CDummy::n << endl;	
return 0;	
}	

In fact, static members have the same properties as global variables but they enjoy class scope. For that reason, and to avoid them to be declared several times, we can only include the prototype (its declaration) in the class declaration but not its definition (its initialization). In order to initialize a static data-member we must include a formal definition outside the class, in the global scope, as in the previous example:

```
int CDummy::n=0;
```

Because it is a unique variable value for all the objects of the same class, it can be referred to as a member of any object of that class or even directly by the class name (of course this is only valid for static members):

```
cout << a.n;
```



```
cout << CDummy::n;
```

These two calls included in the previous example are referring to the same variable: the static variable `n` within class `CDummy` shared by all objects of this class.

Once again, I remind you that in fact it is a global variable. The only difference is its name and possible access restrictions outside its class.

Just as we may include static data within a class, we can also include static functions. They represent the same: they are global functions that are called as if they were object members of a given class. They can only refer to static data, in no case to non-static members of the class, as well as they do not allow the use of the keyword `this`, since it makes reference to an object pointer and these functions in fact are not members of any object but direct members of the class.



Friendship and inheritance

Friend functions

In principle, private and protected members of a class cannot be accessed from outside the same class in which they are declared. However, this rule does not affect *friends*.

Friends are functions or classes declared as such.

If we want to declare an external function as friend of a class, thus allowing this function to have access to the private and protected members of this class, we do it by declaring a prototype of this external function within the class, and preceding it with the keyword `friend`:

// friend functions	24
#include <iostream>	
using namespace std;	
class CRectangle {	
int width, height;	
public:	
void set_values (int, int);	
int area () {return (width * height);}	
friend CRectangle duplicate (CRectangle);	
};	
void CRectangle::set_values (int a, int b) {	
width = a;	
height = b;	
}	
CRectangle duplicate (CRectangle rectparam)	
{	
CRectangle rectres;	
rectres.width = rectparam.width*2;	
rectres.height = rectparam.height*2;	
return (rectres);	
}	

int main () {	
CRectangle rect, rectb;	
rect.set_values (2,3);	
rectb = duplicate (rect);	
cout << rectb.area();	
return 0;	
}	

The duplicate function is a friend of CRectangle. From within that function we have been able to access the members width and height of different objects of type CRectangle, which are private members. Notice that neither in the declaration of duplicate() nor in its later use in main() have we considered duplicate a member of class CRectangle. It isn't! It simply has access to its private and protected members without being a member.

The friend functions can serve, for example, to conduct operations between two different classes. Generally, the use of friend functions is out of an object-oriented programming methodology, so whenever possible it is better to use members of the same class to perform operations with them. Such as in the previous example, it would have been shorter to integrate duplicate() within the class CRectangle.



Friend classes

Just as we have the possibility to define a friend function, we can also define a class as friend of another one, granting that first class access to the protected and private members of the second one.

// friend class	16
#include <iostream>	
using namespace std;	
class CSquare;	
class CRectangle {	
int width, height;	
public:	
int area ()	
{return (width * height);}	
void convert (CSquare a);	
};	
class CSquare {	
private:	
int side;	
public:	
void set_side (int a)	
{side=a;}	
friend class CRectangle;	
};	
void CRectangle::convert (CSquare a) {	
width = a.side;	
height = a.side;	
}	
int main () {	
CSquare sqr;	
CRectangle rect;	
sqr.set_side(4);	

rect.convert(sqr);	
cout << rect.area();	
return 0;	
}	

In this example, we have declared CRectangle as a friend of CSquare so that CRectangle member functions could have access to the protected and private members of CSquare, more concretely to CSquare::side, which describes the side width of the square.

You may also see something new at the beginning of the program: an empty declaration of class CSquare. This is necessary because within the declaration of CRectangle we refer to CSquare (as a parameter in convert()). The definition of CSquare is included later, so if we did not include a previous empty declaration for CSquare this class would not be visible from within the definition of CRectangle.

Consider that friendships are not corresponded if we do not explicitly specify so. In our example, CRectangle is considered as a friend class by CSquare, but CRectangle does not consider CSquare to be a friend, so CRectangle can access the protected and private members of CSquare but not the reverse way. Of course, we could have declared also CSquare as friend of CRectangle if we wanted to.

Another property of friendships is that they are *not transitive*: The friend of a friend is not considered to be a friend unless explicitly specified.

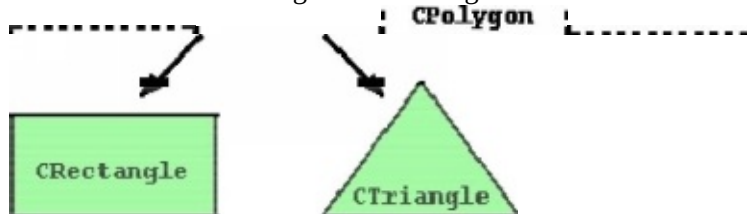
Inheritance between classes

A key feature of C++ classes is inheritance. Inheritance allows to create classes which are derived from other classes, so that they automatically include some of its "parent's" members, plus its own. For example, we are going



to suppose that we want to declare a series of classes that describe polygons like our CRectangle, or like CTriangle. They have certain common properties, such as both can be described by means of only two sides: height and base.

This could be represented in the world of classes with a class CPolygon from which we would derive the two other ones: CRectangle and CTriangle.



The class CPolygon would contain members that are common for both types of polygon. In our case: width and height. And CRectangle and CTriangle would be its derived classes, with specific features that are different from one type of polygon to the other.

Classes that are derived from others inherit all the accessible members of the base class. That means that if a base class includes a member A and we derive it to another class with another member called B, the derived class will contain both members A and B.

In order to derive a class from another, we use a colon (:) in the declaration of the derived class using the following format:

```
class derived_class_name: public base_class_name
{ /*...*/ };
```

Where derived_class_name is the name of the derived class and base_class_name is the name of the class on which it is based. The public access specifier may be replaced by any one of the other access specifiers protected and private. This access specifier describes the minimum access level for the members that are inherited from the base class.



// derived classes	20
#include <iostream>	10
using namespace std;	
class CPolygon {	
protected:	
int width, height;	
public:	
void set_values (int a, int b)	
{ width=a; height=b;}	
};	
class CRectangle: public CPolygon {	
public:	
int area ()	
{ return (width * height); }	
};	
class CTriangle: public CPolygon {	
public:	
int area ()	
{ return (width * height / 2); }	
};	
int main () {	
CRectangle rect;	
CTriangle trgl;	
rect.set_values (4,5);	
trgl.set_values (4,5);	
cout << rect.area() << endl;	
cout << trgl.area() << endl;	
return 0;	
}	

The objects of the classes CRectangle and CTriangle each contain members inherited from CPolygon. These are:
width, height and set_values().

The protected access specifier is similar to private. Its only difference occurs in fact with inheritance. When a class inherits from another one, the members of the derived class can access the protected members inherited from the base class, but not its private members.

Since we wanted width and height to be accessible from members of the derived classes CRectangle and CTriangle and not only by members of CPolygon, we have used protected access instead of private.

We can summarize the different access types according to who can access them in the following way:

Access	public	protected	private
members of the same class	yes	yes	yes
members of derived classes	yes	yes	no
not members	yes	no	no

Where "not members" represent any access from outside the class, such as from main(), from another class or from a function.

In our example, the members inherited by CRectangle and CTriangle have the same access permissions as they had in their base class CPolygon:



CPolygon::width	// protected access
CRectangle::width	// protected access
CPolygon::set_values()	// public access
CRectangle::set_values()	// public access

This is because we have used the public keyword to define the inheritance relationship on each of the derived classes:

```
class CRectangle: public CPolygon { ... }
```

This public keyword after the colon (:) denotes the maximum access level for all the members inherited from the class that follows it (in this case CPolygon). Since public is the most accessible level, by specifying this keyword the derived class will inherit all the members with the same levels they had in the base class.

If we specify a more restrictive access level like protected, all public members of the base class are inherited as protected in the derived class. Whereas if we specify the most restricting of all access levels: private, all the base class members are inherited as private.

For example, if daughter was a class derived from mother that we defined as:

```
class daughter: protected mother;
```

This would set protected as the maximum access level for the members of daughter that it inherited from mother. That is, all members that were public in mother would become protected in daughter. Of course, this would not restrict daughter to declare its own public members. That maximum access level is only set for the members inherited from mother.

If we do not explicitly specify any access level for the inheritance, the compiler assumes private for classes declared with class keyword and public for those declared with struct.

What is inherited from the base class?

In principle, a derived class inherits every member of a base class except:

- its constructor and its destructor

- its operator=() members

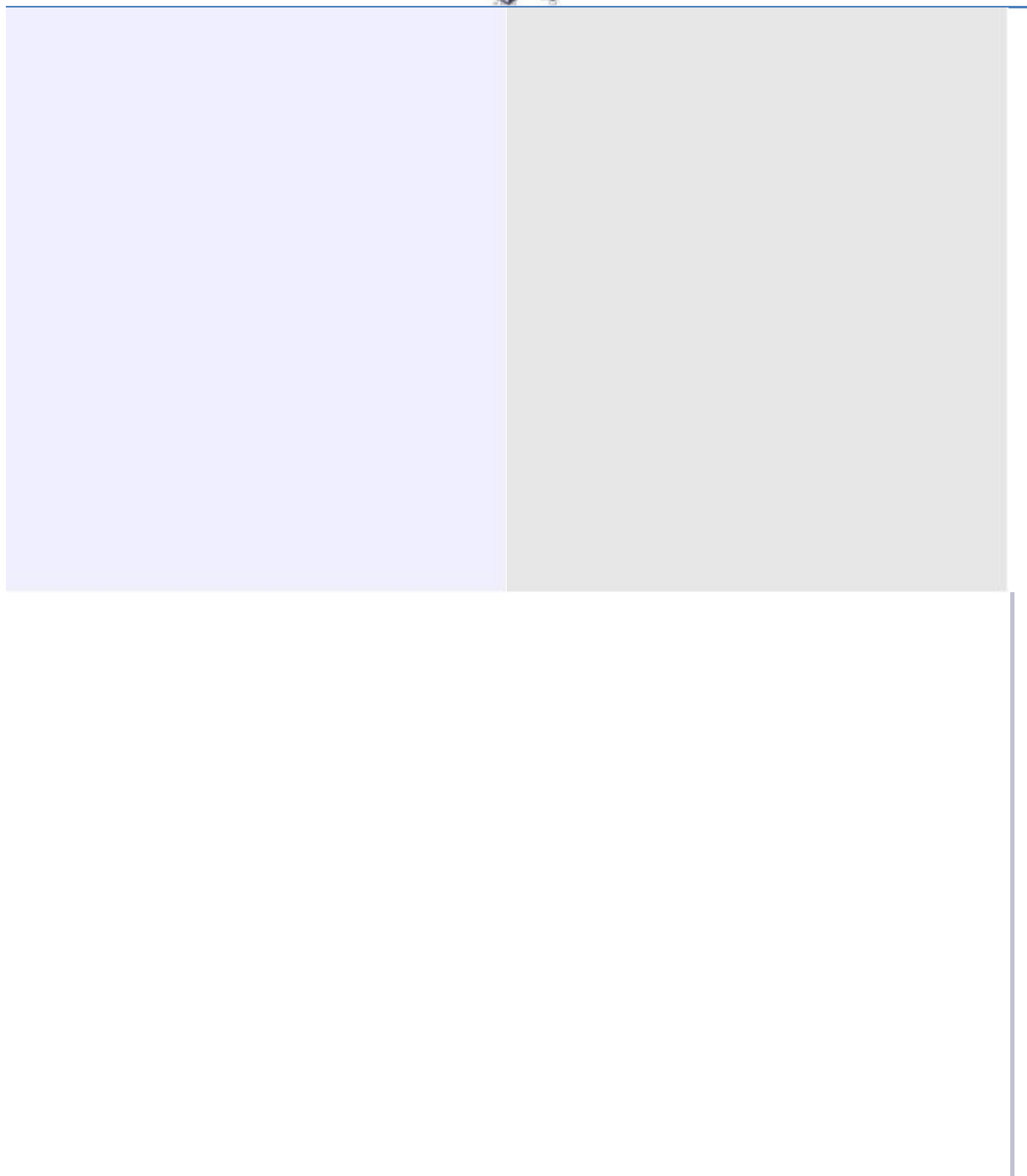
- its friends

Although the constructors and destructors of the base class are not inherited themselves, its default constructor (i.e., its constructor with no parameters) and its destructor are always called when a new object of a derived class is created or destroyed.

If the base class has no default constructor or you want that an overloaded constructor is called when a new derived object is created, you can specify it in each constructor definition of the derived class:

```
derived_constructor_name (parameters) : base_constructor_name (parameters) {...}
```

For example:



```
// constructors and derived classes
#include <iostream>
```

```
using namespace std;
```

```
class mother {
public:
```

```
    mother ()
```

```
    { cout << "mother: no parameters\n"; } mother (int a)
```

```
    { cout << "mother: int parameter\n"; }
```

```
};
```

```
class daughter : public mother {
```

```
public:
```

```
    daughter (int a)
```

```
    { cout << "daughter: int parameter\n\n"; }
```

```
};
```

```
class son : public mother {
```

mother: no parameters
daughter: int parameter

mother: int parameter

son: int parameter

```

        public:
            son (int a) : mother (a)

                { cout << "son: int parameter\n\n"; }

};

int main () {

    daughter cynthia (0);
    son daniel(0);

    return 0;

}

```

Notice the difference between which mother's constructor is called when a new daughter object is created and which when it is a son object. The difference is because the constructor declaration of daughter and son:

```

        daughter (int a)           // nothing specified: call default

        son (int a) : mother (a)   // constructor specified: call this

```

Multiple inheritance

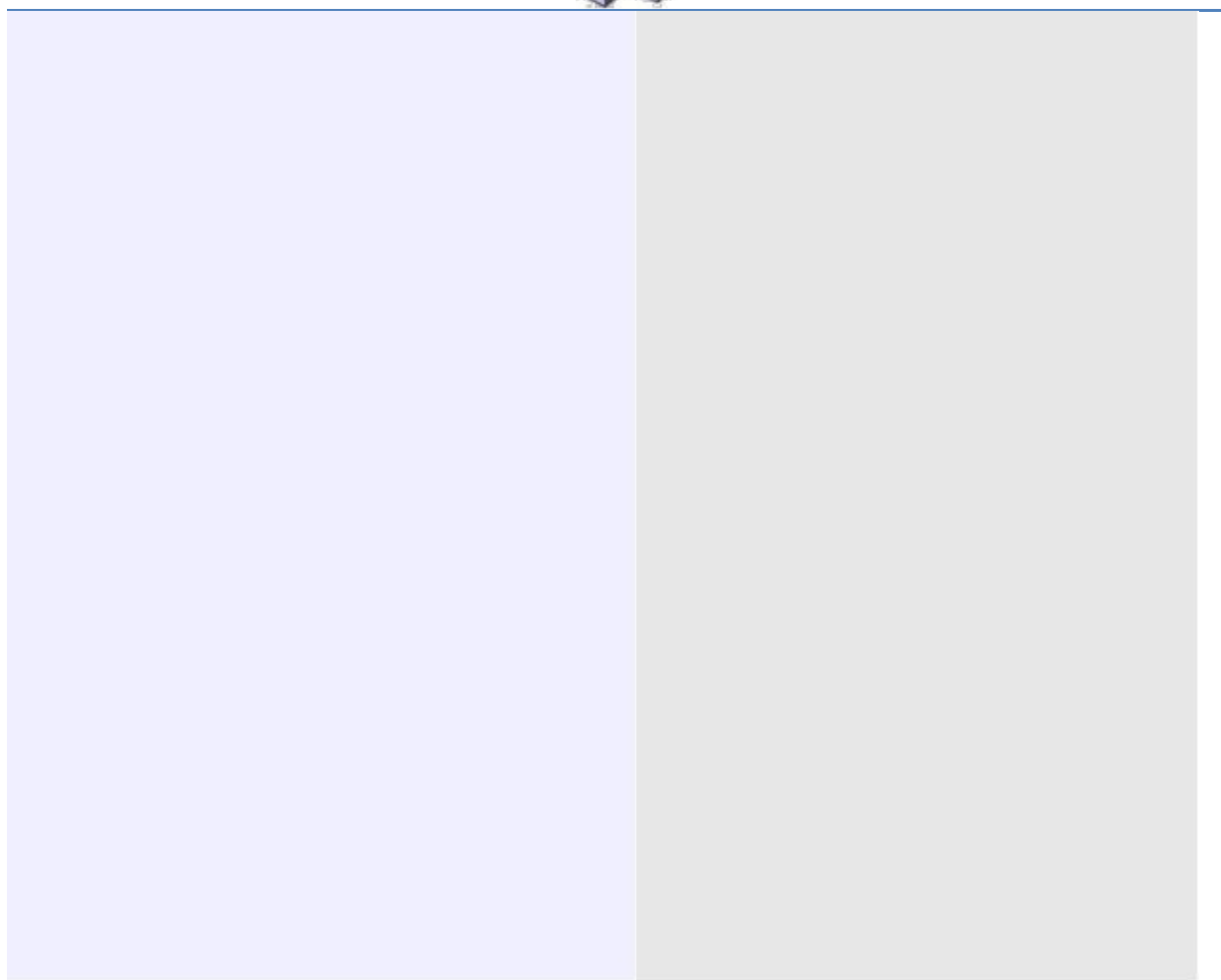
In C++ it is perfectly possible that a class inherits members from more than one class. This is done by simply separating the different base classes with commas in the derived class declaration. For example, if we had a specific class to print on screen (COutput) and we wanted our classes CRectangle and CTriangle to also inherit its members in addition to those of CPolygon we could write:

```

class CRectangle: public CPolygon, public COutput;
class CTriangle: public CPolygon, public COutput;

```

here is the complete example:



// multiple inheritance	20
#include <iostream>	10
using namespace std;	
class CPolygon {	
protected:	
int width, height;	
public:	
void set_values (int a, int b)	
{ width=a; height=b;}	
};	
class COutput {	
public:	
void output (int i);	

};	
void COutput::output (int i) {	
cout << i << endl;	
}	

```
class CRectangle: public CPolygon, public COutput { public:
```

```
    int area ()
    { return (width * height); }
```

```
};
```

```
class CTriangle: public CPolygon, public COutput { public:
```

```
    int area ()
    { return (width * height / 2); }
```

```
};
```

```
int main () {
```

```
    CRectangle rect;
    CTriangle trgl;
```

```
    rect.set_values (4,5);
    trgl.set_values (4,5);
```

```
    rect.output (rect.area());
    trgl.output (trgl.area());
```

```
    return 0;
```

```
}
```




Polymorphism

Before getting into this section, it is recommended that you have a proper understanding of pointers and class inheritance. If any of the following statements seem strange to you, you should review the indicated sections:

Statement:	Explained in:
<code>int a::b(c) {};</code>	Classes
<code>a->b</code>	Data Structures

class a: public b; Friendship and inheritance

Pointers to base class

One of the key features of derived classes is that a pointer to a derived class is type-compatible with a pointer to its base class. Polymorphism is the art of taking advantage of this simple but powerful and versatile feature, that brings Object Oriented Methodologies to its full potential.

We are going to start by rewriting our program about the rectangle and the triangle of the previous section taking into consideration this pointer compatibility property:

<code>// pointers to base class</code>	20
<code>#include <iostream></code>	10
<code>using namespace std;</code>	
<code>class CPolygon {</code>	
<code>protected:</code>	
<code>int width, height;</code>	
<code>public:</code>	
<code>void set_values (int a, int b)</code>	
<code>{ width=a; height=b; }</code>	
<code>};</code>	
<code>class CRectangle: public CPolygon {</code>	
<code>public:</code>	

<code>int area ()</code>	
<code>{ return (width * height); }</code>	
<code>};</code>	
<code>class CTriangle: public CPolygon {</code>	
<code>public:</code>	
<code>int area ()</code>	
<code>{ return (width * height / 2); }</code>	
<code>};</code>	
<code>int main () {</code>	
<code>CRectangle rect;</code>	
<code>CTriangle trgl;</code>	
<code>CPolygon * ppoly1 = &rect;</code>	
<code>CPolygon * ppoly2 = &trgl;</code>	
<code>ppoly1->set_values (4,5);</code>	
<code>ppoly2->set_values (4,5);</code>	
<code>cout << rect.area() << endl;</code>	
<code>cout << trgl.area() << endl;</code>	
<code>return 0;</code>	
<code>}</code>	

In function main, we create two pointers that point to objects of class CPolygon(ppoly1 and ppoly2). Then we assign references to rect and trgl to these pointers, and because both are objects of classes derived from CPolygon, both are valid assignment operations.

The only limitation in using *ppoly1 and *ppoly2 instead of rect and trgl is that both *ppoly1 and *ppoly2 are of type CPolygon* and therefore we can only use these pointers to refer to the members that CRectangle and



CTriangle inherit from CPolygon. For that reason when we call the area() members at the end of the program we have had to use directly the objects rect and trgl instead of the pointers *ppoly1 and *ppoly2.

In order to use area() with the pointers to class CPolygon, this member should also have been declared in the class CPolygon, and not only in its derived classes, but the problem is that CRectangle and CTriangle implement different versions of area, therefore we cannot implement it in the base class. This is when virtual members become handy:

Virtual members

A member of a class that can be redefined in its derived classes is known as a virtual member. In order to declare a member of a class as virtual, we must precede its declaration with the keyword virtual:



// virtual members	20
#include <iostream>	10
using namespace std;	0

```
class CPolygon {  
    protected:  
        int width, height;  
  
    public:  
        void set_values (int a, int b)  
            { width=a; height=b; } virtual int area ()  
            { return (0); }  
};  
  
class CRectangle: public CPolygon {  
    public:  
        int area ()
```

```

        { return (width * height); }
    };

class CTriangle: public CPolygon {
public:
    int area ()
    { return (width * height / 2); }
};

int main () {
    CRectangle rect;
    CTriangle trgl;

    CPolygon poly;

    CPolygon * ppoly1 = &rect;
    CPolygon * ppoly2 = &trgl;

    CPolygon * ppoly3 = &poly;
    ppoly1->set_values (4,5);

    ppoly2->set_values (4,5);
    ppoly3->set_values (4,5);

    cout << ppoly1->area() << endl;
    cout << ppoly2->area() << endl;

    cout << ppoly3->area() << endl;
    return 0;
}

```

Now the three classes (CPolygon, CRectangle and CTriangle) have all the same members: width, height, set_values() and area().

The member function area() has been declared as virtual in the base class because it is later redefined in each derived class. You can verify if you want that if you remove this virtual keyword from the declaration of area() within CPolygon, and then you run the program the result will be 0 for the three polygons instead of 20, 10 and 0. That is because instead of calling the corresponding area() function for each object (CRectangle::area(), CTriangle::area() and CPolygon::area(), respectively), CPolygon::area() will be called in all cases since the calls are via a pointer whose type is CPolygon*.



Therefore, what the virtual keyword does is to allow a member of a derived class with the same name as one in the base class to be appropriately called from a pointer, and more precisely when the type of the pointer is a pointer to the base class but is pointing to an object of the derived class, as in the above example.

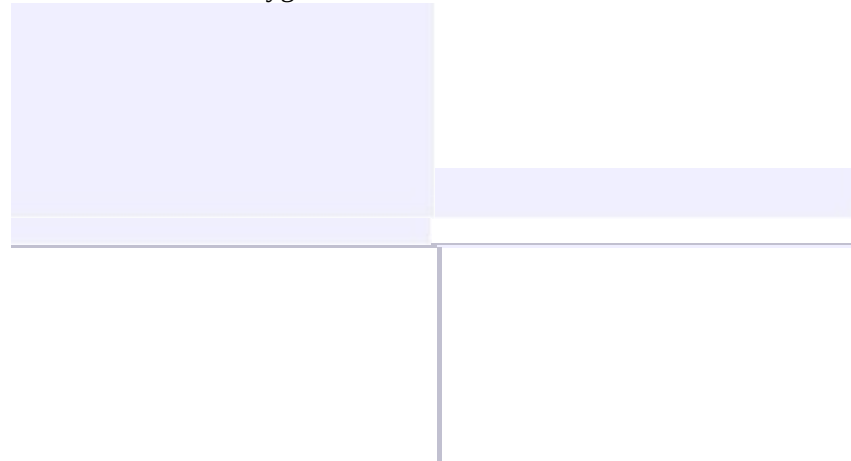
A class that declares or inherits a virtual function is called a *polymorphic class*.

Note that despite of its virtuality, we have also been able to declare an object of type CPolygon and to call its own area() function, which always returns 0.

Abstract base classes

Abstract base classes are something very similar to our CPolygon class of our previous example. The only difference is that in our previous example we have defined a valid area() function with a minimal functionality for objects that were of class CPolygon (like the object poly), whereas in an abstract base classes we could leave that area() member function without implementation at all. This is done by appending =0 (equal to zero) to the function declaration.

An abstract base CPolygon class could look like this:



```
abstract class CPolygon {  
protected:
```

```
    int width, height; public:
```

```
        void set_values (int a, int b) { width=a; height=b; }
```

```
        virtual int area () =0;
```

```
};
```

Notice how we appended `=0` to virtual int area `()` instead of specifying an implementation for the function. This type of function is called a *pure virtual function*, and all classes that contain at least one pure virtual function are *abstract base classes*.

The main difference between an abstract base class and a regular polymorphic class is that because in abstract base classes at least one of its members lacks implementation we cannot create instances (objects) of it.

But a class that cannot instantiate objects is not totally useless. We can create pointers to it and take advantage of all its polymorphic abilities. Therefore a declaration like:

```
CPolygon poly;
```

would not be valid for the abstract base class we have just declared, because tries to instantiate an object. Nevertheless, the following pointers:

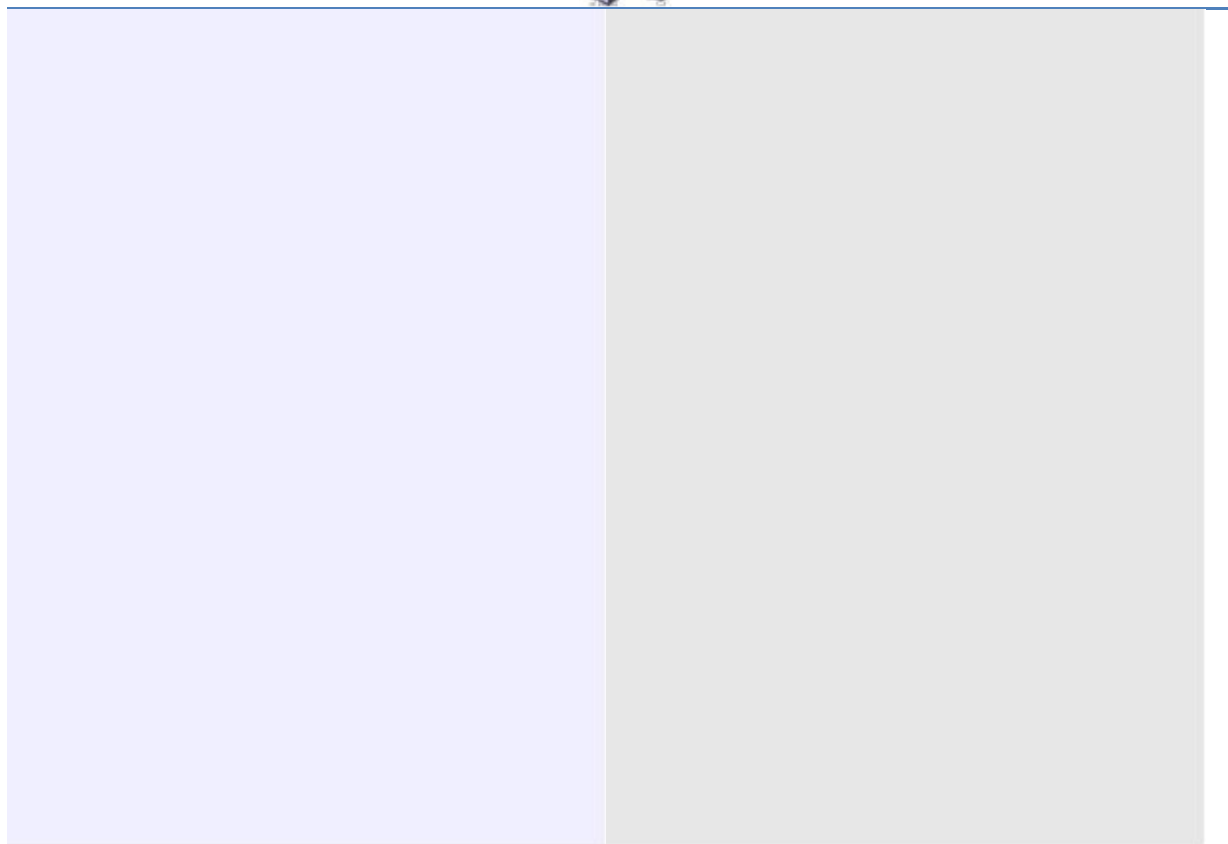
```
CPolygon * ppoly1;
```

```
CPolygon * ppoly2;
```

would be perfectly valid.

This is so for as long as `CPolygon` includes a pure virtual function and therefore it's an abstract base class. However, pointers to this abstract base class can be used to point to objects of derived classes.

Here you have the complete example:



// abstract base class	20
#include <iostream>	10
using namespace std;	

```

class CPolygon {
    protected:

        int width, height;

    public:
        void set_values (int a, int b)
            { width=a; height=b; } virtual int area (void) =0;

};

class CRectangle: public CPolygon {
    public:

        int area (void)
            { return (width * height); }

};

class CTriangle: public CPolygon {
    public:

        int area (void)
            { return (width * height / 2); }

};

int main () {
    CRectangle rect;

    CTriangle trgl;
    CPolygon * ppoly1 = &rect;

    CPolygon * ppoly2 = &trgl;
    ppoly1->set_values (4,5);

    ppoly2->set_values (4,5);

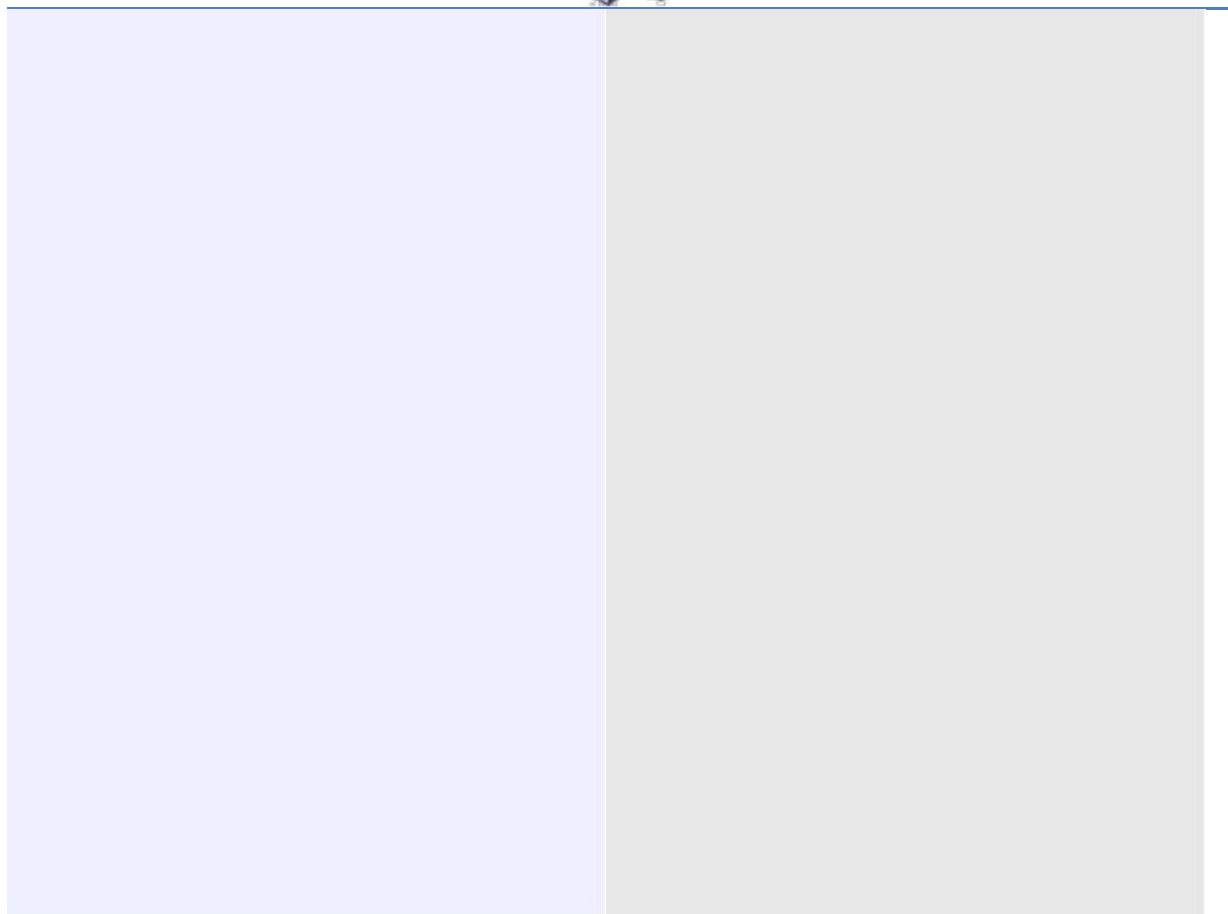
    cout << ppoly1->area() << endl;
    cout << ppoly2->area() << endl;

    return 0;
}

```

```
}
```

If you review the program you will notice that we refer to objects of different but related classes using a unique type of pointer (CPolygon*). This can be tremendously useful. For example, now we can create a function member of the abstract base class CPolygon that is able to print on screen the result of the area() function even though CPolygon itself has no implementation for this function:



// pure virtual members can be called	20
// from the abstract base class	10
#include <iostream>	
using namespace std;	

```

class CPolygon {
    protected:
        int width, height;
    public:
        void set_values (int a, int b)
        { width=a; height=b; } virtual int area (void) =0; void printarea (void)
        { cout << this->area() << endl; }
};

```

```

class CRectangle: public CPolygon {
    public:
        int area (void)
            { return (width * height); }
};

class CTriangle: public CPolygon {
    public:
        int area (void)
            { return (width * height / 2); }
};

int main () {
    CRectangle rect;
    CTriangle trgl;

    CPolygon * ppoly1 = &rect;

    CPolygon * ppoly2 = &trgl;
    ppoly1->set_values (4,5);

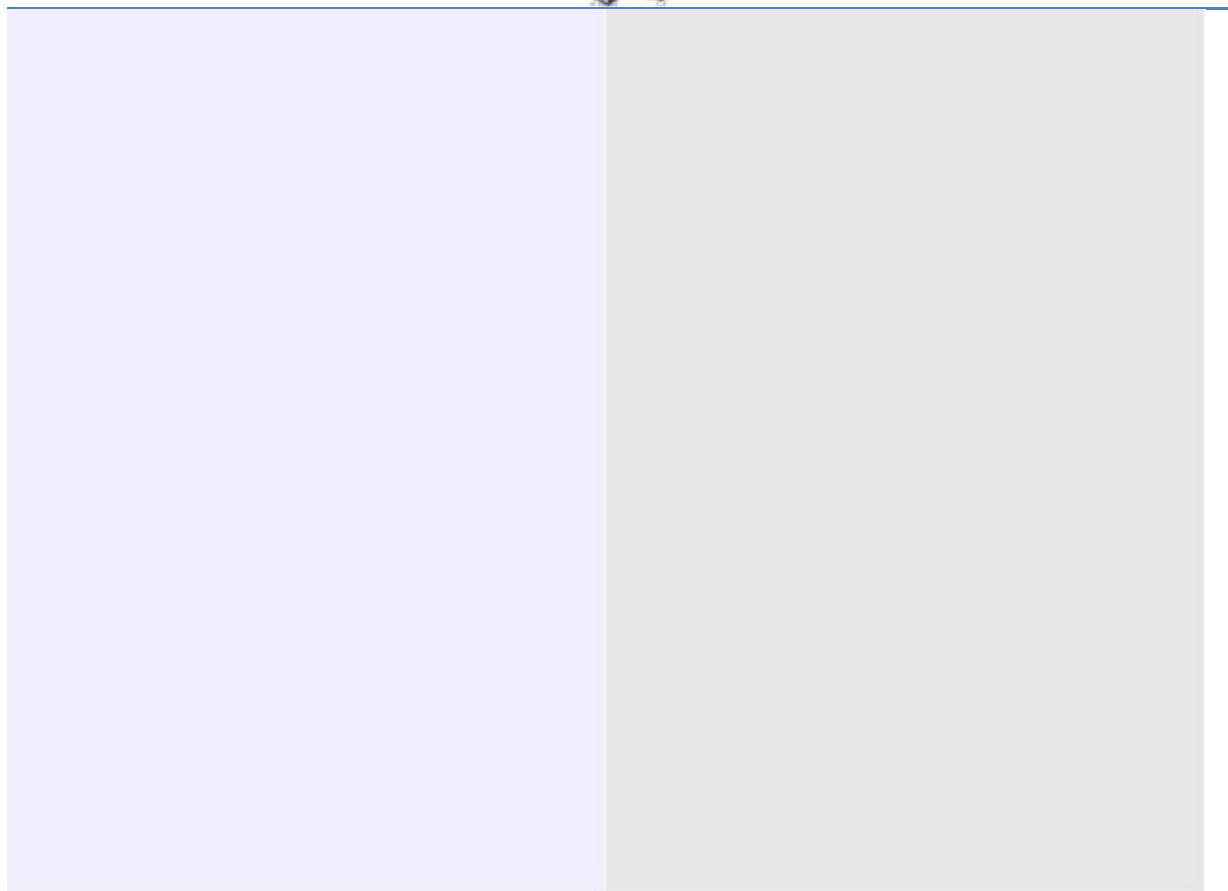
    ppoly2->set_values (4,5);
    ppoly1->printarea();

    ppoly2->printarea();
    return 0;
}

```

Virtual members and abstract classes grant C++ the polymorphic characteristics that make object-oriented programming such a useful instrument in big projects. Of course, we have seen very simple uses of these features, but these features can be applied to arrays of objects or dynamically allocated objects.

Let's end with the same example again, but this time with objects that are dynamically allocated:



// dynamic allocation and polymorphism	20
#include <iostream>	10
using namespace std;	

```
class CPolygon {
    protected:

        int width, height;

    public:
        void set_values (int a, int b)

            { width=a; height=b; } virtual int area (void) =0; void printarea (void)
            { cout << this->area() << endl; }

};

class CRectangle: public CPolygon {
    public:
```



```

        int area (void)
            { return (width * height); }

};

class CTriangle: public CPolygon {
public:

    int area (void)
        { return (width * height / 2); }

};

int main () {
    CPolygon * ppoly1 = new CRectangle;

    CPolygon * ppoly2 = new CTriangle;
    ppoly1->set_values (4,5);

    ppoly2->set_values (4,5);

    ppoly1->printarea();
    ppoly2->printarea();

    delete ppoly1;
    delete ppoly2;

    return 0;
}

```

Notice that the ppoly pointers:



```

CPolygon * ppoly1 = new CRectangle;
CPolygon * ppoly2 = new CTriangle;

```

are declared being of type pointer to CPolygon but the objects dynamically allocated have been declared having the derived class type directly.



Advanced concepts

Templates

Function templates

Function templates are special functions that can operate with *generic types*. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

In C++ this can be achieved using *template parameters*. A template parameter is a special kind of parameter that can be used to pass a type as argument: just like regular function parameters can be used to pass values to a function, template parameters allow to pass also types to a function. These function templates can use these parameters as if they were any other regular type.

The format for declaring function templates with type parameters is:

```
template <class identifier> function_declaration; template <typename identifier> function_declaration;
```

The only difference between both prototypes is the use of either the keyword `class` or the keyword `typename`. Its use is indistinct, since both expressions have exactly the same meaning and behave exactly the same way.

For example, to create a template function that returns the greater one of two objects we could use:

```
template <class myType>
myType GetMax (myType a, myType b) {
    return (a>b?a:b);
}
```

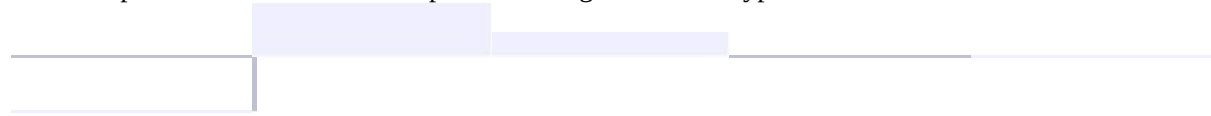
Here we have created a template function with `myType` as its template parameter. This template parameter represents a type that has not yet been specified, but that can be used in the template function as if it were a regular type. As you can see, the function template `GetMax` returns the greater of two parameters of this

still-undefined type.

To use this function template we use the following format for the function call:

function_name <type> (parameters);

For example, to call GetMax to compare two integer values of type int we can write:



```
int x,y;  
GetMax <int> (x,y);
```

When the compiler encounters this call to a template function, it uses the template to automatically generate a function replacing each appearance of myType by the type passed as the actual template parameter (int in this case) and then calls it. This process is automatically performed by the compiler and is invisible to the programmer.

Here is the entire example:



// function template	6
#include <iostream>	10
using namespace std;	
template <class T>	
T GetMax (T a, T b) {	
T result;	
result = (a>b)? a : b;	
return (result);	
}	
int main () {	
int i=5, j=6, k;	
long l=10, m=5, n;	
k=GetMax<int>(i,j);	
n=GetMax<long>(l,m);	
cout << k << endl;	
cout << n << endl;	
return 0;	
}	

In this case, we have used T as the template parameter name instead of myType because it is shorter and in fact is a very common template parameter name. But you can use any identifier you like.

In the example above we used the function template GetMax() twice. The first time with arguments of type int and the second one with arguments of type long. The compiler has instantiated and then called each time the appropriate version of the function.

As you can see, the type T is used within the GetMax() template function even to declare new objects of that type:

```
T result;
```

Therefore, result will be an object of the same type as the parameters a and b when the function template is instantiated with a specific type.

In this specific case where the generic type T is used as a parameter for GetMax the compiler can find out automatically which data type has to instantiate without having to explicitly specify it within angle brackets (like we have done before specifying <int> and <long>). So we could have written instead:



```
int i,j;
```

```
GetMax (i,j);
```

Since both i and j are of type int, and the compiler can automatically find out that the template parameter can only be int. This implicit method produces exactly the same result:



// function template II	6
#include <iostream>	10
using namespace std;	
template <class T>	
T GetMax (T a, T b) {	
return (a>b?a:b);	
}	
int main () {	
int i=5, j=6, k;	
long l=10, m=5, n;	
k=GetMax(i,j);	
n=GetMax(l,m);	
cout << k << endl;	
cout << n << endl;	
return 0;	
}	

Notice how in this case, we called our function template GetMax() without explicitly specifying the type between angle-brackets <>. The compiler automatically determines what type is needed on each call.

Because our template function includes only one template parameter (class T) and the function template itself accepts two parameters, both of this T type, we cannot call our function template with two objects of different types as arguments:

```
int i;  
long l;  
k = GetMax (i,l);
```

This would not be correct, since our GetMax function template expects two arguments of the same type, and in this call to it we use objects of two different types.

We can also define function templates that accept more than one type parameter, simply by specifying more template parameters between the angle brackets. For example:

```
template <class T, class U>
T GetMin (T a, U b) {

    return (a<b?a:b);

}
```

In this case, our function template GetMin() accepts two parameters of different types and returns an object of the same type as the first parameter (T) that is passed. For example, after that declaration we could call GetMin() with:

```
int i,j;
long l;

i = GetMin<int,long> (j,l);
```

or simply:

```
i = GetMin (j,l);
```

even though j and l have different types, since the compiler can determine the appropriate instantiation anyway.



Class templates

We also have the possibility to write class templates, so that a class can have members that use template parameters as types. For example:

```
template <class T>
class mypair {
```

```
    T values [2];
```

```
    public:
```

```
    mypair (T first, T second)
```

```
    {
```

```
        values[0]=first; values[1]=second;
```

```
    }
```

```
};
```

The class that we have just defined serves to store two elements of any valid type. For example, if we wanted to declare an object of this class to store two integer values of type `int` with the values 115 and 36 we would write:

```
mypair<int> myobject (115, 36);
```

this same class would also be used to create an object to store any other type:

```
mypair<double> myfloats (3.0, 2.18);
```

The only member function in the previous class template has been defined inline within the class declaration itself. In case that we define a function member outside the declaration of the class template, we must always precede that definition with the template <...> prefix:

// class	templates	100
#include <iostream>		
using namespace std;		
template <class T>		
class mypair {		
T a, b;		
public:		
mypair (T first, T second)		
{a=first; b=second;}		
T getmax ();		
};		
template <class T>		
T mypair<T>::getmax ()		
{		
T retval;		
retval = a>b? a : b;		
return retval;		
}		
int main () {		
mypair <int> myobject (100, 75);		
cout << myobject.getmax();		
return 0;		
}		

Notice the syntax of the definition of member function getmax:



```
template <class T>
```

```
T mypair<T>::getmax ()
```

Confused by so many T's? There are three T's in this declaration: The first one is the template parameter. The second T refers to the type returned by the function. And the third T (the one between angle brackets) is also a requirement: It specifies that this function's template parameter is also the class template parameter.

Template specialization

If we want to define a different implementation for a template when a specific type is passed as template parameter, we can declare a specialization of that template.

For example, let's suppose that we have a very simple class called mycontainer that can store one element of any type and that it has just one member function called increase, which increases its value. But we find that when it stores an element of type char it would be more convenient to have a completely different implementation with a function member uppercase, so we decide to declare a class template specialization for that type:

// template specialization	8
#include <iostream>	J
using namespace std;	

```

class template: template <class T> class mycontainer {
    T element; public:
        mycontainer (T arg) {element=arg;} T increase () {return ++element;}
};

class template specialization: template <>
    class mycontainer <char> { char element;
        public:
            mycontainer (char arg) {element=arg;} char uppercase ()
            {
                if ((element>='a')&&(element<='z')) element+='A'-'a';
                return element;
            }
};

```

```
int main () {  
  
    mycontainer<int> myint (7);  
    mycontainer<char> mychar ('j');  
  
    cout << myint.increase() << endl;  
    cout << mychar.uppercase() << endl;  
  
    return 0;  
}
```

This is the syntax used in the class template specialization:

```
template <> class mycontainer <char> { ... };
```

First of all, notice that we precede the class template name with an empty `template<>` parameter list. This is to explicitly declare it as a template specialization.



But more important than this prefix, is the `<char>` specialization parameter after the class template name. This specialization parameter itself identifies the type for which we are going to declare a template class specialization (`char`). Notice the differences between the generic class template and the specialization:

template	<class T> class mycontainer	{	}	;
template	<> class mycontainer <char>	...{	}	;

The first line is the generic template, and the second one is the specialization.

When we declare specializations for a template class, we must also define all its members, even those exactly equal to the generic template class, because there is no "inheritance" of members from the generic template to the specialization.

Non-type parameters for templates

Besides the template arguments that are preceded by the `class` or `typename` keywords, which represent types, templates can also have regular typed parameters, similar to those found in functions. As an example, have a look at this class template that is used to contain sequences of elements:



// sequence template	100
#include <iostream>	3.1416
using namespace std;	
template <class T, int N>	

<code>class mysequence {</code>	
<code> T memblock [N];</code>	
<code>public:</code>	
<code> void setmember (int x, T value);</code>	
<code> T getmember (int x);</code>	
<code>};</code>	

```
template <class T, int N>
```

```
void mysequence<T,N>::setmember (int x, T value)
```

```
{
```

```
    memblock[x]=value;
}
```

```
template <class T, int N>
```

```
T mysequence<T,N>::getmember (int x) {
```

```
    return memblock[x];
}
```

```
int main () {
```

```
    mysequence <int,5> myints;
    mysequence <double,5> myfloats;
```

```
    myints.setmember (0,100);
    myfloats.setmember (3,3.1416);
```

```
    cout << myints.getmember(0) << '\n';
    cout << myfloats.getmember(3) << '\n';
```

```
    return 0;
}
```

It is also possible to set default values or types for class template parameters. For example, if the previous class template definition had been:

```
template <class T=char, int N=10> class mysequence {..};
```

We could create objects using the default template parameters by declaring:

```
mysequence<> myseq;
```

Which would be equivalent to:



```
mysequence<char,10> myseq;
```

Templates and multiple-file projects

From the point of view of the compiler, templates are not normal functions or classes. They are compiled on demand, meaning that the code of a template function is not compiled until an instantiation with specific template arguments is required. At that moment, when an instantiation is required, the compiler generates a function specifically for those arguments from the template.

When projects grow it is usual to split the code of a program in different source code files. In these cases, the interface and implementation are generally separated. Taking a library of functions as example, the interface generally consists of declarations of the prototypes of all the functions that can be called. These are generally declared in a "header file" with a .h extension, and the implementation (the definition of these functions) is in an independent file with c++ code.

Because templates are compiled when required, this forces a restriction for multi-file projects: the implementation (definition) of a template class or function must be in the same file as its declaration. That means that we cannot separate the interface in a separate header file, and that we must include both interface and implementation in any file that uses the templates.

Since no code is generated until a template is instantiated when required, compilers are prepared to allow the inclusion more than once of the same template file with both declarations and definitions in a project without generating linkage errors.



Namespaces

Namespaces allow to group entities like classes, objects and functions under a name. This way the global scope can be divided in "sub-scopes", each one with its own name.

The format of namespaces is:

namespace identifier

```
{  
entities  
}
```

Where identifier is any valid identifier and entities is the set of classes, objects and functions that are included within the namespace. For example:

```
namespace myNamespace  
{  
  
    int a, b;  
}
```

In this case, the variables a and b are normal variables declared within a namespace called myNamespace. In order to access these variables from outside the myNamespace namespace we have to use the scope operator ::. For example, to access the previous variables from outside myNamespace we can write:

```
myNamespace::a  
myNamespace::b
```

The functionality of namespaces is especially useful in the case that there is a possibility that a global object or function uses the same identifier as another one, causing redefinition errors. For example:

// namespaces	5
#include <iostream>	3.1416

using namespace std;	
namespace first	
{	
int var = 5;	
}	
namespace second	
{	
double var = 3.1416;	
}	
int main () {	
cout << first::var << endl;	
cout << second::var << endl;	
return 0;	
}	

In this case, there are two global variables with the same name: var. One is defined within the namespace first and the other one in second. No redefinition errors happen thanks to namespaces.



using

The keyword using is used to introduce a name from a namespace into the current declarative region. For

example:	
// using	5
#include <iostream>	2.7183
using namespace std;	10
	3.1416
namespace first	
{	
int x = 5;	
int y = 10;	
}	
namespace second	
{	
double x = 3.1416;	
double y = 2.7183;	
}	
int main () {	
using first::x;	
using second::y;	
cout << x << endl;	
cout << y << endl;	
cout << first::y << endl;	
cout << second::x << endl;	
return 0;	
}	

Notice how in this code, x (without any name qualifier) refers to first::x whereas y refers to second::y, exactly as our using declarations have specified. We still have access to first::y and second::x using their fully qualified names.

The keyword using can also be used as a directive to introduce an entire namespace:

// using	5
#include <iostream>	10
using namespace std;	3.1416
	2.7183
namespace first	
{	
int x = 5;	
int y = 10;	
}	
namespace second	
{	
double x = 3.1416;	
double y = 2.7183;	
}	
int main () {	
using namespace first;	
cout << x << endl;	
cout << y << endl;	
cout << second::x << endl;	
cout << second::y << endl;	
return 0;	
}	

In this case, since we have declared that we were using namespace first, all direct uses of x and y without name qualifiers were referring to their declarations in namespace first.



using and using namespace have validity only in the same block in which they are stated or in the entire code if they are used directly in the global scope. For example, if we had the intention to first use the objects of one namespace and then those of another one, we could do something like:

// using namespace example	5
#include <iostream>	3.1416
using namespace std;	
namespace first	
{	
int x = 5;	
}	
namespace second	
{	
double x = 3.1416;	
}	
int main () {	
{	
using namespace first;	
cout << x << endl;	
}	
{	
using namespace second;	
cout << x << endl;	
}	
return 0;	
}	

Namespace alias

We can declare alternate names for existing namespaces according to the following format:

```
namespace new_name = current_name;
```

Namespace std

All the files in the C++ standard library declare all of its entities within the std namespace. That is why we have generally included the `using namespace std;` statement in all programs that used any entity defined in `iostream`.



Exceptions

Exceptions provide a way to react to exceptional circumstances (like runtime errors) in our program by transferring control to special functions called *handlers*.

To catch exceptions we must place a portion of code under exception inspection. This is done by enclosing that portion of code in a *try block*. When an exceptional circumstance arises within that block, an exception is thrown that transfers the control to the exception handler. If no exception is thrown, the code continues normally and all handlers are ignored.

A exception is thrown by using the `throw` keyword from inside the `try` block. Exception handlers are declared with the keyword `catch`, which must be placed immediately after the `try` block:



```
// exceptions
#include <iostream>

using namespace std;

int main () {
    try
    {
        throw 20;
    }
    catch (int e)
    {
        cout << "An exception occurred. "
        cout << "Exception Nr. " << e << endl;
    }

    return 0;
}
```

An exception occurred. Exception Nr. 20

The code under exception handling is enclosed in a try block. In this example this code simply throws an exception:

```
throw 20;
```

A throw expression accepts one parameter (in this case the integer value 20), which is passed as an argument to the exception handler.

The exception handler is declared with the catch keyword. As you can see, it follows immediately the closing brace of the try block. The catch format is similar to a regular function that always has at least one parameter. The type of this parameter is very important, since the type of the argument passed by the throw expression is checked against it, and only in the case they match, the exception is caught.

We can chain multiple handlers (catch expressions), each one with a different parameter type. Only the handler that matches its type with the argument specified in the throw statement is executed.

If we use an ellipsis (...) as the parameter of catch, that handler will catch any exception no matter what the type of the throw exception is. This can be used as a default handler that catches all exceptions not caught by other handlers if it is specified at last:

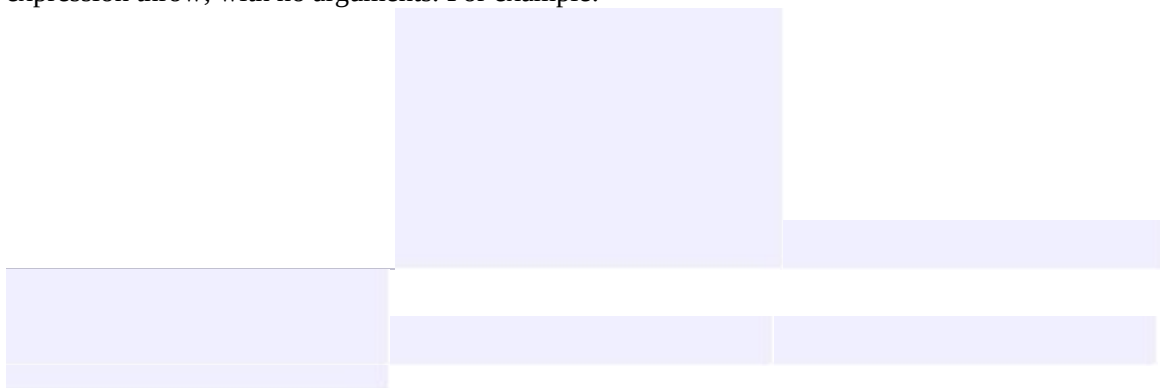


```
try {  
  
    // code here  
  
}  
  
catch (int param) { cout << "int exception"; }  
  
catch (char param) { cout << "char exception"; }  
  
catch (...) { cout << "default exception"; }
```

In this case the last handler would catch any exception thrown with any parameter that is neither an int nor a char.

After an exception has been handled the program execution resumes after the try-catch block, not after the throw statement!.

It is also possible to nest try-catch blocks within more external try blocks. In these cases, we have the possibility that an internal catch block forwards the exception to its external level. This is done with the expression `throw;` with no arguments. For example:



```

try {

    try {

        // code here

    }
    catch (int n) {

        throw;

    }

}
catch (...) {

    cout << "Exception occurred";

}

```

Exception specifications

When declaring a function we can limit the exception type it might directly or indirectly throw by appending a throw suffix to the function declaration:

```
float myfunction (char param) throw (int);
```

This declares a function called myfunction which takes one argument of type char and returns an element of type float. The only exception that this function might throw is an exception of type int. If it throws an exception with a different type, either directly or indirectly, it cannot be caught by a regular int-type handler.

If this throw specifier is left empty with no type, this means the function is not allowed to throw exceptions. Functions with no throw specifier (regular functions) are allowed to throw exceptions with any type:

int	myfunction	(int	param) throw(); //	no exceptions allowed
int	myfunction	(int	param); //	all exceptions allowed

Standard exceptions

The C++ Standard library provides a base class specifically designed to declare objects to be thrown as exceptions. It is called exception and is defined in the <exception> header file under the namespace std. This class has the usual default and copy constructors, operators and destructors, plus an additional virtual member function called what that returns a null-terminated character sequence (char *) and that can be

overwritten in derived classes to contain some sort of description of the exception.



// standard exceptions	My exception happened.
#include <iostream>	
#include <exception>	
using namespace std;	
class myexception: public exception	
{	
virtual const char* what() const throw()	
{	
return "My exception happened";	
}	
} myex;	
int main () {	
try	
{	
throw myex;	
}	
catch (exception& e)	
{	
cout << e.what() << endl;	
}	
return 0;	
}	

We have placed a handler that catches exception objects by reference (notice the ampersand & after the type), therefore this catches also classes derived from exception, like our myex object of class myexception.

All exceptions thrown by components of the C++ Standard library throw exceptions derived from this std::exception class. These are:

exception	description
bad_alloc	thrown by new on allocation failure
bad_cast	thrown by dynamic_cast when fails with a referenced type
bad_exception	thrown when an exception type doesn't match any catch
bad_typeid	thrown by typeid

ios_base::failure thrown by functions in the iostream library

For example, if we use the operator new and the memory cannot be allocated, an exception of type bad_alloc is thrown:

```
try
```

```
{
```

```
    int * myarray= new int[1000];
```

```
}
```

```
catch (bad_alloc&)
```

```
{
```

```
    cout << "Error allocating memory." << endl;
```

```
}
```

It is recommended to include all dynamic memory allocations within a try block that catches this type of exception to perform a clean action instead of an abnormal program termination, which is what happens when this type of exception is thrown and not caught. If you want to force a bad_alloc exception to see it in action, you can try to allocate a huge array; On my system, trying to allocate 1 billion ints threw a bad_alloc exception.

Because bad_alloc is derived from the standard base class exception, we can handle that same exception by catching references to the exception class:



```
bad_alloc standard exception #include <iostream>
#include <exception> using namespace std;
int main () {
    try
    {
```

```
        int* myarray= new int[1000];  
    }  
    catch (exception& e)  
    {  
        cout << "Standard exception: " << e.what()  
        endl;  
    }  
    return 0;  
}
```




Type Casting

Converting an expression of a given type into another type is known as *type-casting*. We have already seen some ways to type cast:

Implicit conversion

Implicit conversions do not require any operator. They are automatically performed when a value is copied to a compatible type. For example:

```
short a=2000;  
int b;  
  
b=a;
```

Here, the value of a has been promoted from short to int and we have not had to specify any type-casting operator. This is known as a standard conversion. Standard conversions affect fundamental data types, and allow conversions such as the conversions between numerical types (short to int, int to float, double to int...), to or from bool, and some pointer conversions. Some of these conversions may imply a loss of precision, which the compiler can signal with a warning. This can be avoided with an explicit conversion.

Implicit conversions also include constructor or operator conversions, which affect classes that include specific constructors or operator functions to perform conversions. For example:

```
class A {};  
  
class B { public: B (A a) {} };  
  
    a;  
B b=a;
```

Here, an implicit conversion happened between objects of class A and class B, because B has a constructor

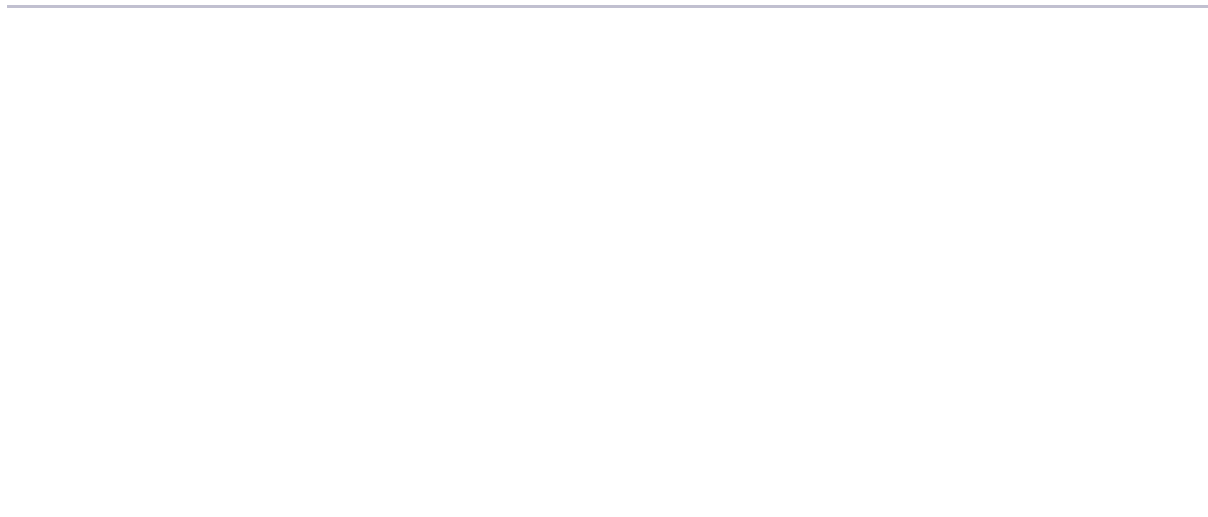
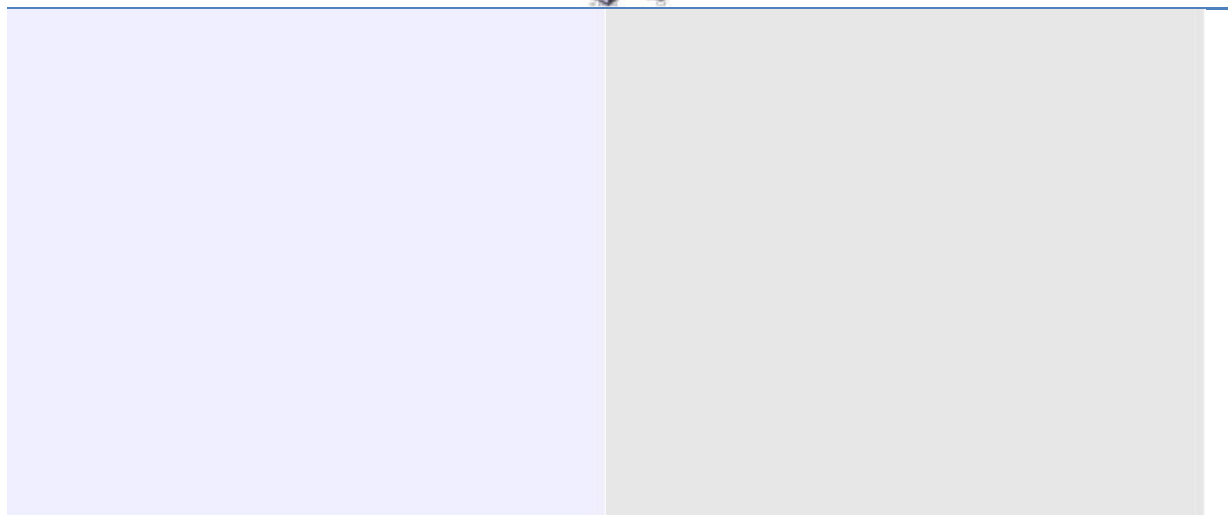
that takes an object of class A as parameter. Therefore implicit conversions from A to B are allowed.

Explicit conversion

C++ is a strong-typed language. Many conversions, specially those that imply a different interpretation of the value, require an explicit conversion. We have already seen two notations for explicit type conversion: functional and c-like casting:

<code>short a=2000;</code>	
<code>int b;</code>	
<code>b = (int) a;</code>	<code>// c-like cast notation</code>
<code>b = int (a);</code>	<code>// functional notation</code>

The functionality of these explicit conversion operators is enough for most needs with fundamental data types. However, these operators can be applied indiscriminately on classes and pointers to classes, which can lead to code that while being syntactically correct can cause runtime errors. For example, the following code is syntactically correct:



```

class type-casting #include <iostream> using namespace std;

class CDummy {
    float i,j;
};

class CAddition {
    int x,y;
public:
    CAddition (int a, int b) { x=a; y=b; }
    int result() { return x+y;}
};

int main () {
    CDummy d;

    CAddition * padd;
    padd = (CAddition*) &d;

    cout << padd->result();
    return 0;
}

```

The program declares a pointer to CAddition, but then it assigns to it a reference to an object of another incompatible type using explicit type-casting:

```
padd = (CAddition*) &d;
```

Traditional explicit type-casting allows to convert any pointer into any other pointer type, independently of the types they point to. The subsequent call to member result will produce either a run-time error or a unexpected result.

In order to control these types of conversions between classes, we have four specific casting operators:

dynamic_cast, reinterpret_cast, static_cast and const_cast. Their format is to follow the new type enclosed between angle-brackets (<>) and immediately after, the expression to be converted between parentheses.

```
dynamic_cast <new_type> (expression)
reinterpret_cast <new_type> (expression)
```

```
static_cast <new_type> (expression)
```

```
const_cast <new_type> (expression)
```

The traditional type-casting equivalents to these expressions would be:

`(new_type) expression`
`new_type (expression)`

but each one with its own special characteristics:

dynamic_cast

`dynamic_cast` can be used only with pointers and references to objects. Its purpose is to ensure that the result of the type conversion is a valid complete object of the requested class.

Therefore, `dynamic_cast` is always successful when we cast a class to one of its base classes:



```
class CBase { };
```

```
class CDerived: public CBase { };
```

```
CBase b; CBase* pb;
```

```
CDerived d; CDerived* pd;
```

pb	=	<code>dynamic_cast<CBase*>(&d);</code>	//	ok: derived-to-base
pd	=	<code>dynamic_cast<CDerived*>(&b);</code>	//	wrong: base-to-derived

The second conversion in this piece of code would produce a compilation error since base-to-derived conversions are not allowed with `dynamic_cast` unless the base class is polymorphic.

When a class is polymorphic, `dynamic_cast` performs a special checking during runtime to ensure that the expression yields a valid complete object of the requested class:



// dynamic_cast

Null pointer on second type-

```

#include <iostream>
#include <exception>

using namespace std;

class CBase { virtual void dummy() {} };
class CDerived: public CBase { int a; };

int main () {

    try {

        CBase * pba = new CDerived;

        CBase * pbb = new CBase;
        CDerived * pd;

        pd = dynamic_cast<CDerived*>(pba);

        if (pd==0) cout << "Null pointer on first type-cast" << endl;

        pd = dynamic_cast<CDerived*>(pbb);
        if (pd==0) cout << "Null pointer on second type-cast" << endl;

    } catch (exception& e) {cout << "Exception: " << e.what();} return 0;

}

```

Compatibility note: dynamic_cast requires the Run-Time Type Information (RTTI) to keep track of dynamic types. Some compilers support this feature as an option which is disabled by default. This must be enabled for runtime type checking using dynamic_cast to work properly.

The code tries to perform two dynamic casts from pointer objects of type CBase*(pba and pbb) to a pointer object of type CDerived*, but only the first one is successful. Notice their respective initializations:

```

CBase * pba = new CDerived;
CBase * pbb = new CBase;

```

Even though both are pointers of type CBase*, pba points to an object of type CDerived, while pbb points to an object of type CBase. Thus, when their respective type-castings are performed using dynamic_cast, pba is pointing to a full object of class CDerived, whereas pbb is pointing to an object of class CBase, which is an incomplete object of class CDerived.

When dynamic_cast cannot cast a pointer because it is not a complete object of the required class -as in the second conversion in the previous example- it returns a null pointer to indicate the failure. If dynamic_cast is used to convert to a reference type and the conversion is not possible, an exception of type bad_cast is

thrown instead.

`dynamic_cast` can also cast null pointers even between pointers to unrelated classes, and can also cast pointers of any type to void pointers (`void*`).



static_cast

`static_cast` can perform conversions between pointers to related classes, not only from the derived class to its base, but also from a base class to its derived. This ensures that at least the classes are compatible if the proper object is converted, but no safety check is performed during runtime to check if the object being converted is in fact a full object of the destination type. Therefore, it is up to the programmer to ensure that the conversion is safe. On the other side, the overhead of the type-safety checks of `dynamic_cast` is avoided.

```
class CBase {};  
class CDerived: public CBase {};  
  
CBase * a = new CBase;  
CDerived * b = static_cast<CDerived*>(a);
```

This would be valid, although `b` would point to an incomplete object of the class and could lead to runtime errors if dereferenced.

`static_cast` can also be used to perform any other non-pointer conversion that could also be performed implicitly, like for example standard conversion between fundamental types:

```
double d=3.14159265;  
int i = static_cast<int>(d);
```

Or any conversion between classes with explicit constructors or operator functions as described in "implicit conversions" above.

reinterpret_cast

`reinterpret_cast` converts any pointer type to any other pointer type, even of unrelated classes. The operation result is a simple binary copy of the value from one pointer to the other. All pointer conversions are allowed: neither the content pointed nor the pointer type itself is checked.

It can also cast pointers to or from integer types. The format in which this integer value represents a pointer is platform-specific. The only guarantee is that a pointer cast to an integer type large enough to fully contain it, is granted to be able to be cast back to a valid pointer.

The conversions that can be performed by `reinterpret_cast` but not by `static_cast` have no specific uses in C++ are low-level operations, whose interpretation results in code which is generally system-specific, and thus non-portable. For example:

```
class A {};  
class B {};  
  
A * a = new A;  
B * b = reinterpret_cast<B*>(a);
```

This is valid C++ code, although it does not make much sense, since now we have a pointer that points to an object of an incompatible class, and thus dereferencing it is unsafe.

const_cast

This type of casting manipulates the constness of an object, either to be set or to be removed. For example, in order to pass a const argument to a function that expects a non-constant parameter:



// const_cast	sample text
#include <iostream>	
using namespace std;	
void print (char * str)	
{	
cout << str << endl;	
}	
int main () {	
const char * c = "sample text";	
print (const_cast<char *> (c));	
return 0;	
}	

typeid

typeid allows to check the type of an expression:

typeid (expression)

This operator returns a reference to a constant object of type `type_info` that is defined in the standard header file `<typeinfo>`. This returned value can be compared with another one using operators `==` and `!=` or can serve to obtain a null-terminated character sequence representing the data type or class name by using its `name()` member.

// typeid	a and b are of different types:
#include <iostream>	a is: int *
#include <typeinfo>	b is: int
using namespace std;	
int main () {	
int * a,b;	
a=0; b=0;	
if (typeid(a) != typeid(b))	
{	
cout << "a and b are of different types:\n";	
cout << "a is: " << typeid(a).name() << '\n';	

cout << "b is: " << typeid(b).name() << '\n';	
}	
return 0;	
}	

When typeid is applied to classes typeid uses the RTTI to keep track of the type of dynamic objects. When typeid is applied to an expression whose type is a polymorphic class, the result is the type of the most derived complete object:



// typeid, polymorphic class	a is: class CBase *
#include <iostream>	b is: class CBase *
#include <typeinfo>	*a is: class CBase
#include <exception>	*b is: class CDerived
using namespace std;	
class CBase { virtual void f(){} };	
class CDerived : public CBase {};	
int main () {	
try {	
CBase* a = new CBase;	
CBase* b = new CDerived;	
cout << "a is: " << typeid(a).name() << '\n';	
cout << "b is: " << typeid(b).name() << '\n';	
cout << "*a is: " << typeid(*a).name() << '\n';	
cout << "*b is: " << typeid(*b).name() << '\n';	
} catch (exception& e) { cout << "Exception: " << e.what() << endl;	
}	
return 0;	
}	

Notice how the type that typeid considers for pointers is the pointer type itself (both a and b are of type class CBase *). However, when typeid is applied to objects (like *a and *b) typeid yields their dynamic type (i.e. the type of their most derived complete object).

If the type typeid evaluates is a pointer preceded by the dereference operator (*), and this pointer has a null value, typeid throws a bad_typeid exception.



Preprocessor directives

Preprocessor directives are lines included in the code of our programs that are not program statements but directives for the preprocessor. These lines are always preceded by a hash sign (#). The preprocessor is executed before the actual compilation of code begins, therefore the preprocessor digests all these directives before any code is generated by the statements.

These preprocessor directives extend only across a single line of code. As soon as a newline character is found, the preprocessor directive is considered to end. No semicolon (;) is expected at the end of a preprocessor directive. The only way a preprocessor directive can extend through more than one line is by preceding the newline character at the end of the line by a backslash (\).

macro definitions (#define, #undef)

To define preprocessor macros we can use #define. Its format is:

#define identifier replacement

When the preprocessor encounters this directive, it replaces any occurrence of identifier in the rest of the code by replacement. This replacement can be an expression, a statement, a block or simply anything. The preprocessor does not understand C++, it simply replaces any occurrence of identifier by replacement.

```
#define TABLE_SIZE 100
```

```
int table1[TABLE_SIZE];  
int table2[TABLE_SIZE];
```

After the preprocessor has replaced TABLE_SIZE, the code becomes equivalent to:

```
int table1[100];
```

```
int table2[100];
```

This use of #define as constant definer is already known by us from previous tutorials, but #define can work also with parameters to define function macros:

`#define getmax(a,b) a>b?a:b`

This would replace any occurrence of `getmax` followed by two arguments by the replacement expression, but also replacing each argument by its identifier, exactly as you would expect if it was a function:

<code>// function macro</code>	5
<code>#include <iostream></code>	7
<code>using namespace std;</code>	
<code>#define getmax(a,b) ((a)>(b)?(a):(b))</code>	
<code>int main()</code>	
<code>{</code>	
<code> int x=5, y;</code>	
<code> y= getmax(x,2);</code>	
<code> cout << y << endl;</code>	
<code> cout << getmax(7,x) << endl;</code>	
<code> return 0;</code>	
<code>}</code>	

Defined macros are not affected by block structure. A macro lasts until it is undefined with the `#undef` preprocessor directive:



```
#define TABLE_SIZE 100
```

```
int table1[TABLE_SIZE];  
#undef TABLE_SIZE
```

```
#define TABLE_SIZE 200  
int table2[TABLE_SIZE];
```

This would generate the same code as:

```
int table1[100];  
int table2[200];
```

Function macro definitions accept two special operators (# and ##) in the replacement sequence:

If the operator # is used before a parameter is used in the replacement sequence, that parameter is replaced by a string literal (as if it were enclosed between double quotes)

```
#define str(x) #x
```

```
cout << str(test);
```

This would be translated into:

```
cout << "test";
```

The operator ## concatenates two arguments leaving no blank spaces between them:

```
#define glue(a,b) a ## b
```

```
glue(c,out) << "test";
```

This would also be translated into:

```
cout << "test";
```

Because preprocessor replacements happen before any C++ syntax check, macro definitions can be a tricky feature, but be careful: code that relies heavily on complicated macros may result obscure to other programmers, since the syntax they expect is on many occasions different from the regular expressions programmers expect in C++.

Conditional inclusions (#ifdef, #ifndef, #if, #endif, #else and #elif)

These directives allow to include or discard part of the code of a program if a certain condition is met.

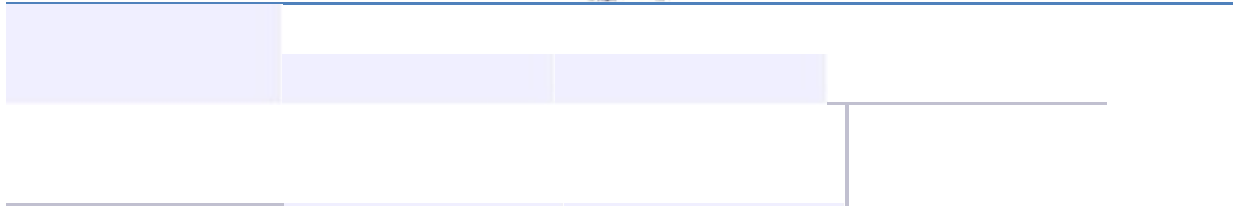
#ifdef allows a section of a program to be compiled only if the macro that is specified as the parameter has been defined, no matter which its value is. For example:

```
#ifdef TABLE_SIZE
int table[TABLE_SIZE];

#endif
```

In this case, the line of code `int table[TABLE_SIZE];` is only compiled if `TABLE_SIZE` was previously defined with `#define`, independently of its value. If it was not defined, that line will not be included in the program compilation.

#ifndef serves for the exact opposite: the code between `#ifndef` and `#endif` directives is only compiled if the specified identifier has not been previously defined. For example:



```
#ifndef TABLE_SIZE
```

```
#define TABLE_SIZE 100  
#endif
```

```
int table[TABLE_SIZE];
```

In this case, if when arriving at this piece of code, the `TABLE_SIZE` macro has not been defined yet, it would be defined to a value of 100. If it already existed it would keep its previous value since the `#define` directive would not be executed.

The `#if`, `#else` and `#elif` (i.e., "else if") directives serve to specify some condition to be met in order for the portion of code they surround to be compiled. The condition that follows `#if` or `#elif` can only evaluate constant expressions, including macro expressions. For example:



```

    #if TABLE_SIZE>200

    #undef TABLE_SIZE
    #define TABLE_SIZE 200

    #elif TABLE_SIZE<50

    #undef TABLE_SIZE
    #define TABLE_SIZE 50

    #else

    #undef TABLE_SIZE
    #define TABLE_SIZE 100

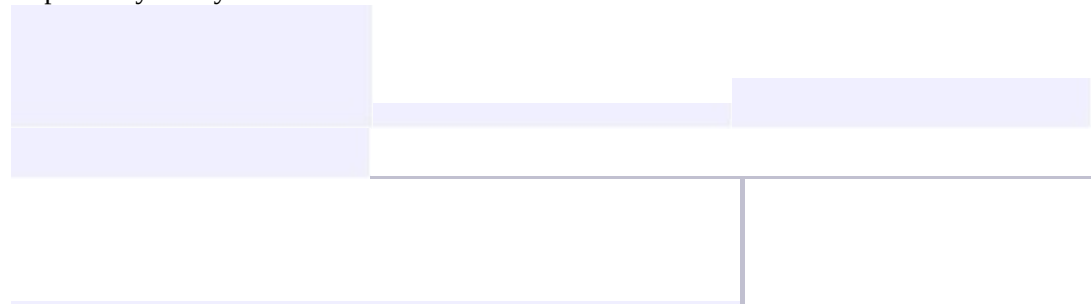
    #endif

    int table[TABLE_SIZE];

```

Notice how the whole structure of #if, #elif and #else chained directives ends with #endif.

The behavior of #ifdef and #ifndef can also be achieved by using the special operators defined and !defined respectively in any #if or #elif directive:



```

    #if !defined TABLE_SIZE
    #define TABLE_SIZE 100

    #elif defined ARRAY_SIZE
    #define TABLE_SIZE ARRAY_SIZE

    int table[TABLE_SIZE];

```

Line control (#line)

When we compile a program and some error happen during the compiling process, the compiler shows an error message with references to the name of the file where the error happened and a line number, so it is easier to find the code generating the error.

The #line directive allows us to control both things, the line numbers within the code files as well as the file name that we want that appears when an error takes place. Its format is:

```
#line number "filename"
```

Where number is the new line number that will be assigned to the next code line. The line numbers of successive lines will be increased one by one from this point on.

"filename" is an optional parameter that allows to redefine the file name that will be shown. For example:



```
#line 20 "assigning variable"
```

```
int a?;
```

This code will generate an error that will be shown as error in file "assigning variable", line 20.

Error directive (#error)

This directive aborts the compilation process when it is found, generating a compilation the error that can be specified as its parameter:

```
#ifndef __cplusplus
```

```
#error A C++ compiler is required!
```

```
#endif
```

This example aborts the compilation process if the macro name `__cplusplus` is not defined (this macro name is defined by default in all C++ compilers).

Source file inclusion (#include)

This directive has also been used assiduously in other sections of this tutorial. When the preprocessor finds an `#include` directive it replaces it by the entire content of the specified file. There are two ways to specify a file to be included:

```
#include "file"
```

```
#include <file>
```

The only difference between both expressions is the places (directories) where the compiler is going to look for the file. In the first case where the file name is specified between double-quotes, the file is searched first in the same directory that includes the file containing the directive. In case that it is not

there, the compiler searches the file in the default directories where it is configured to look for the standard header files.

If the file name is enclosed between angle-brackets `<>` the file is searched directly where the compiler is configured to look for the standard header files. Therefore, standard header files are usually included in angle-brackets, while other specific header files are included using quotes.

Pragma directive (#pragma)

This directive is used to specify diverse options to the compiler. These options are specific for the platform and the compiler you use. Consult the manual or the reference of your compiler for more information on the possible parameters that you can define with `#pragma`.

If the compiler does not support a specific argument for `#pragma`, it is ignored - no error is generated.

Predefined macro names

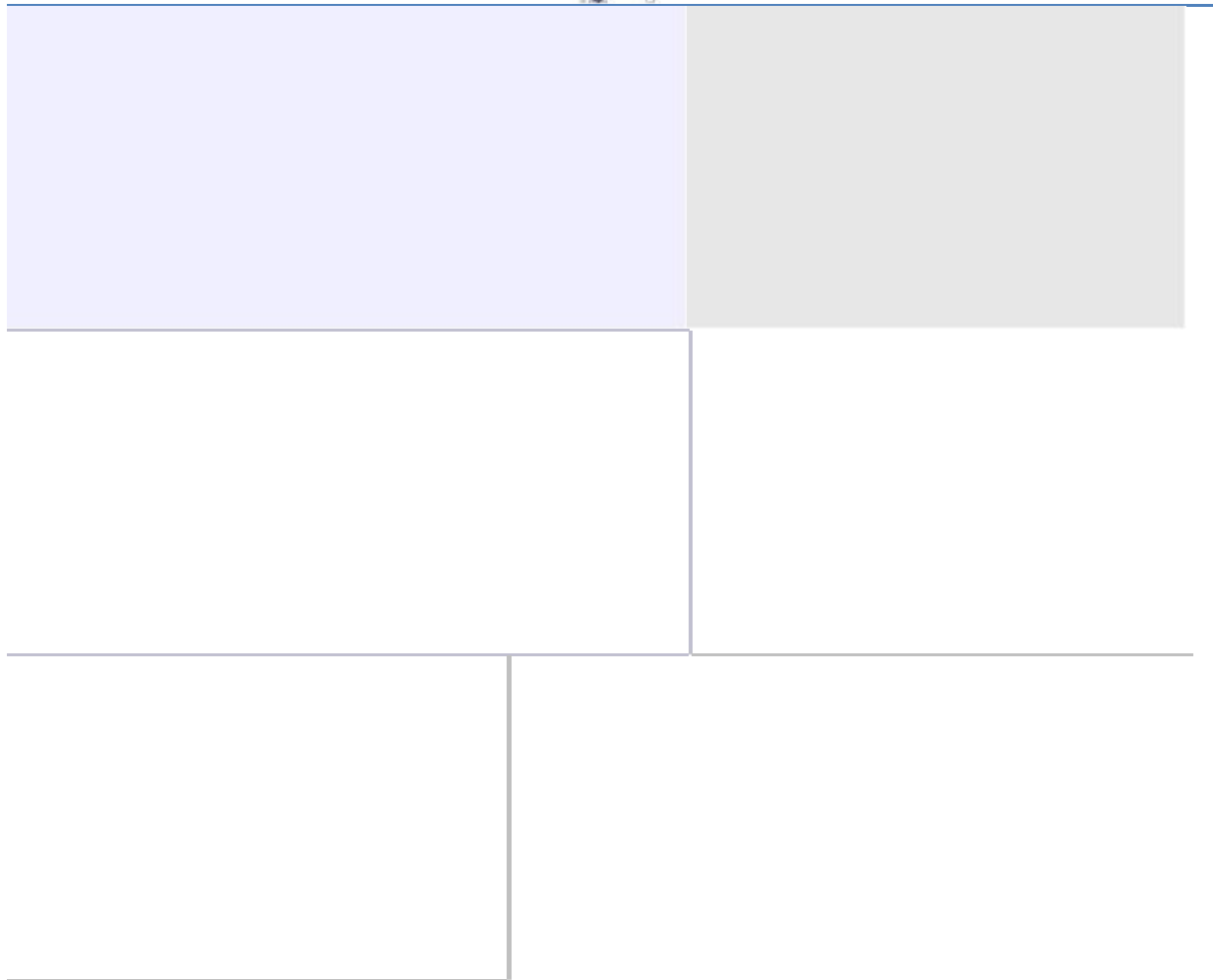
The following macro names are defined at any time:

macro	value
<code>__LINE__</code>	Integer value representing the current line in the source code file being compiled.
<code>__FILE__</code>	A string literal containing the presumed name of the source file being compiled.
<code>__DATE__</code>	A string literal in the form "Mmm dd yyyy" containing the date in which the compilation process began.
<code>__TIME__</code>	A string literal in the form "hh:mm:ss" containing the time at which the compilation process began.

An integer value. All C++ compilers have this constant defined to some value. If the compiler is fully `__cplusplus` compliant with the C++ standard its value is equal or greater than 199711L depending on the version

of the standard they comply.

For example:



standard macro names `#include <iostream> using namespace std;`

```
int main()
{
    cout << "This is the line number " << __LINE__ << " of file " << __FILE__ << ".\n";
    cout << "Its compilation began " << __DATE__ << " at " << __TIME__ << ".\n";

    cout << "The compiler gives a __cplusplus value of "; cout << __cplusplus;
    return 0;
}
```

This is the line number 7 of file /home/jay/stdmacronames.cpp.

Its compilation began Nov 1 2005 at 10:12:29.

The compiler gives a __cplusplus value of 1



C++ Standard Library

Input/Output with files

C++ provides the following classes to perform output and input of characters to/from files:

ofstream: Stream class to write on files

ifstream: Stream class to read from files

fstream: Stream class to both read and write from/to files.

These classes are derived directly or indirectly from the classes istream, and ostream. We have already used objects whose types were these classes: cin is an object of class istream and cout is an object of class ostream. Therefore, we have already been using classes that are related to our file streams. And in fact, we can use our file streams the same way we are already used to use cin and cout, with the only difference that we have to associate these streams with physical files. Let's see an example:

// basic	file operations	[file example.txt]
#include	<iostream>	Writing this to a file
#include <fstream>		
using namespace std;		
int main () {		
ofstream myfile;		
myfile.open ("example.txt");		
myfile << "Writing this to a file.\n";		
myfile.close();		
return 0;		
}		

This code creates a file called example.txt and inserts a sentence into it in the same way we are used to do with cout, but using the file stream myfile instead.

But let's go step by step:

Open a file

The first operation generally performed on an object of one of these classes is to associate it to a real file. This procedure is known as to *open a file*. An open file is represented within a program by a stream object (an instantiation of one of these classes, in the previous example this was `myfile`) and any input or output operation performed on this stream object will be applied to the physical file associated to it.

In order to open a file with a stream object we use its member function `open()`:

```
open (filename, mode);
```

Where `filename` is a null-terminated character sequence of type `const char *` (the same type that string literals have) representing the name of the file to be opened, and `mode` is an optional parameter with a combination of the following flags:



ios::in	Open for input operations.
ios::out	Open for output operations.
ios::binaryOpen in binary mode.	
ios::ate	Set the initial position at the end of the file.
	If this flag is not set to any value, the initial position is the beginning of the file.
ios::app	All output operations are performed at the end of the file, appending the content to the current content
	of the file. This flag can only be used in streams open for output-only operations.

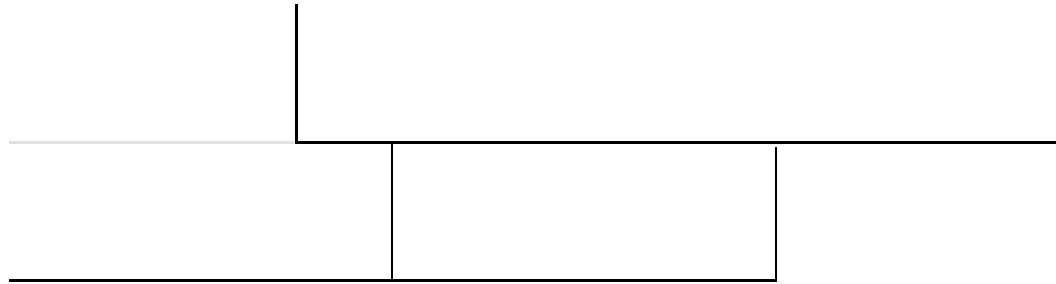
If the file opened for output operations already existed before, its previous content is deleted and `ios::trunc` replaced by the new one.

All these flags can be combined using the bitwise operator OR (`|`). For example, if we want to open the file `example.bin` in binary mode to add data we could do it by the following call to member function `open()`:

```
ofstream myfile;
```

```
myfile.open ("example.bin", ios::out | ios::app | ios::binary);
```

Each one of the `open()` member functions of the classes `ofstream`, `ifstream` and `fstream` has a default mode that is used if the file is opened without a second argument:



class default mode parameter

```
ofstream ios::out
```

```
ifstream ios::in
```

```
fstream ios::in | ios::out
```

For `ifstream` and `ofstream` classes, `ios::in` and `ios::out` are automatically and respectively assumed, even if a mode that does not include them is passed as second argument to the `open()` member function.

The default value is only applied if the function is called without specifying any value for the mode parameter. If the function is called with any value in that parameter the default mode is overridden, not combined.

File streams opened in binary mode perform input and output operations independently of any format considerations. Non-binary files are known as *text files*, and some translations may occur due to formatting of some special characters (like newline and carriage return characters).

Since the first task that is performed on a file stream object is generally to open a file, these three classes include a constructor that automatically calls the `open()` member function and has the exact same parameters as this member. Therefore, we could also have declared the previous `myfile` object and conducted the same opening operation in our previous example by writing:

```
ofstream myfile ("example.bin", ios::out | ios::app | ios::binary);
```

Combining object construction and stream opening in a single statement. Both forms to open a file are valid and equivalent.

To check if a file stream was successful opening a file, you can do it by calling to member `is_open()` with no arguments. This member function returns a `bool` value of `true` in the case that indeed the stream object is associated with an open file, or `false` otherwise:

```
if (myfile.is_open()) { /* ok, proceed with output */ }
```




Closing a file

When we are finished with our input and output operations on a file we shall close it so that its resources become available again. In order to do that we have to call the stream's member function `close()`. This member function takes no parameters, and what it does is to flush the associated buffers and close the file:

```
myfile.close();
```

Once this member function is called, the stream object can be used to open another file, and the file is available again to be opened by other processes.

In case that an object is destructed while still associated with an open file, the destructor automatically calls the member function `close()`.

Text files

Text file streams are those where we do not include the `ios::binary` flag in their opening mode. These files are designed to store text and thus all values that we input or output from/to them can suffer some formatting transformations, which do not necessarily correspond to their literal binary value.

Data output operations on text files are performed in the same way we operated with `cout`:

// writing on a text file	[file example.txt]
#include <iostream>	This is a line.
#include <fstream>	This is another line.
using namespace std;	
int main () {	
ofstream myfile ("example.txt");	
if (myfile.is_open())	
{	
myfile << "This is a line.\n";	
myfile << "This is another line.\n";	
myfile.close();	
}	
else cout << "Unable to open file";	
return 0;	
}	

Data input from a file can also be performed in the same way that we did with cin:



// reading a text file	This is a line.
#include <iostream>	This is another line.
#include <fstream>	
#include <string>	
using namespace std;	
int main () {	
string line;	
ifstream myfile ("example.txt");	
if (myfile.is_open())	
{	
while (! myfile.eof())	
{	
getline (myfile,line);	
cout << line << endl;	
}	
myfile.close();	
}	
else cout << "Unable to open file";	
return 0;	
}	

This last example reads a text file and prints out its content on the screen. Notice how we have used a new member function, called eof() that returns true in the case that the end of the file has been reached. We have created a while loop that finishes when indeed myfile.eof() becomes true (i.e., the end of the file has been reached).

Checking state flags

In addition to eof(), which checks if the end of file has been reached, other member functions exist to check the state of a stream (all of them return a bool value):

bad()

Returns true if a reading or writing operation fails. For example in the case that we try to write to a file that is not open for writing or if the device where we try to write has no

space left.

fail()

Returns true in the same cases as `bad()`, but also in the case that a format error happens, like when an alphabetical character is extracted when we are trying to read an integer number.

eof()

Returns true if a file open for reading has reached the end.

good()

It is the most generic state flag: it returns false in the same cases in which calling any of the previous functions would return true.

In order to reset the state flags checked by any of these member functions we have just seen we can use the member function `clear()`, which takes no parameters.

get and put stream pointers

All i/o streams objects have, at least, one internal stream pointer:

`ifstream`, like `istream`, has a pointer known as the *get pointer* that points to the element to be read in the next input operation.



ofstream, like ostream, has a pointer known as the *put pointer* that points to the location where the next element has to be written.

Finally, fstream, inherits both, the get and the put pointers, from istream (which is itself derived from both istream and ostream).

These internal stream pointers that point to the reading or writing locations within a stream can be manipulated using the following member functions:

tellg() and tellp()

These two member functions have no parameters and return a value of the member type pos_type, which is an integer data type representing the current position of the get stream pointer (in the case of tellg) or the put stream pointer (in the case of tellp).

seekg() and seekp()

These functions allow us to change the position of the get and put stream pointers. Both functions are overloaded with two different prototypes. The first prototype is:

```
seekg ( position );
```

```
seekp ( position );
```

Using this prototype the stream pointer is changed to the absolute position position (counting from the beginning of the file). The type for this parameter is the same as the one returned by functions tellg and tellp: the member type pos_type, which is an integer value.

The other prototype for these functions is:

```
seekg ( offset, direction );
```

```
seekp ( offset, direction );
```

Using this prototype, the position of the get or put pointer is set to an offset value relative to some specific point determined by the parameter direction. offset is of the member type off_type, which is also an integer type. And direction is of type seekdir, which is an enumerated type (enum) that determines the point from where offset is counted from, and that can take any of the following values:

`ios::begoffset` counted from the beginning of the stream

`ios::cur` offset counted from the current position of the stream pointer

`ios::endoffset` counted from the end of the stream

The following example uses the member functions we have just seen to obtain the size of a file:



// obtaining file size	size is: 40 bytes.
#include <iostream>	
#include <fstream>	
using namespace std;	
int main () {	
long begin,end;	
ifstream myfile ("example.txt");	
begin = myfile.tellg();	
myfile.seekg (0, ios::end);	
end = myfile.tellg();	
myfile.close();	
cout << "size is: " << (end-begin) << " bytes.\n";	
return 0;	
}	

Binary files

In binary files, to input and output data with the extraction and insertion operators (<< and >>) and functions like getline is not efficient, since we do not need to format any data, and data may not use the separation codes used by text files to separate elements (like space, newline, etc...).

File streams include two member functions specifically designed to input and output binary data sequentially: write and read. The first one (write) is a member function of ostream inherited by ofstream. And read is a member function of istream that is inherited by ifstream. Objects of class fstream have both members. Their prototypes are:

```
write ( memory_block, size );
```

```
read ( memory_block, size );
```

Where memory_block is of type "pointer to char" (char*), and represents the address of an array of bytes where the read data elements are stored or from where the data elements to be written are taken. The size parameter is an integer value that specifies the number of characters to be read or written from/to the memory block.

// reading a complete binary file	the complete file content is in memory
#include <iostream>	
#include <fstream>	

using namespace std;	
ifstream::pos_type size;	
char * memblock;	
int main () {	
ifstream file ("example.bin",	
ios::in ios::binary ios::ate);	
if (file.is_open())	
{	
size = file.tellg();	
memblock = new char [size];	
file.seekg (0, ios::beg);	
file.read (memblock, size);	
file.close();	
cout << "the complete file content is in memory";	
delete[] memblock;	
}	
else cout << "Unable to open file";	
return 0;	
}	

In this example the entire file is read and stored in a memory block. Let's examine how this is done:



First, the file is open with the `ios::ate` flag, which means that the get pointer will be positioned at the end of the file. This way, when we call to member `tellg()`, we will directly obtain the size of the file. Notice the type we have used to declare variable `size`:

```
ifstream::pos_type size;
```

`ifstream::pos_type` is a specific type used for buffer and file positioning and is the type returned by `file.tellg()`. This type is defined as an integer type, therefore we can conduct on it the same operations we conduct on any other integer value, and can safely be converted to another integer type large enough to contain the size of the file. For a file with a size under 2GB we could use `int`:

```
int size;  
size = (int) file.tellg();
```

Once we have obtained the size of the file, we request the allocation of a memory block large enough to hold the entire file:

```
memblock = new char[size];
```

Right after that, we proceed to set the get pointer at the beginning of the file (remember that we opened the file with this pointer at the end), then read the entire file, and finally close it:

```
file.seekg (0, ios::beg);  
file.read (memblock, size);  
  
file.close();
```

At this point we could operate with the data obtained from the file. Our program simply announces that the content of the file is in memory and then terminates.

Buffers and Synchronization

When we operate with file streams, these are associated to an internal buffer of type `streambuf`. This buffer is a memory block that acts as an intermediary between the stream and the physical file. For example, with an `ofstream`, each time the member function `put` (which writes a single character) is called, the character is not written directly to the physical file with which the stream is associated. Instead of that, the character is inserted in that stream's intermediate buffer.

When the buffer is flushed, all the data contained in it is written to the physical medium (if it is an output stream) or simply freed (if it is an input stream). This process is called *synchronization* and takes place under any of the following circumstances:

When the file is closed: before closing a file all buffers that have not yet been flushed are synchronized and all pending data is written or read to the physical medium.

When the buffer is full: Buffers have a certain size. When the buffer is full it is automatically synchronized.

Explicitly, with manipulators: When certain manipulators are used on streams, an explicit synchronization takes place. These manipulators are: `flush` and `endl`.

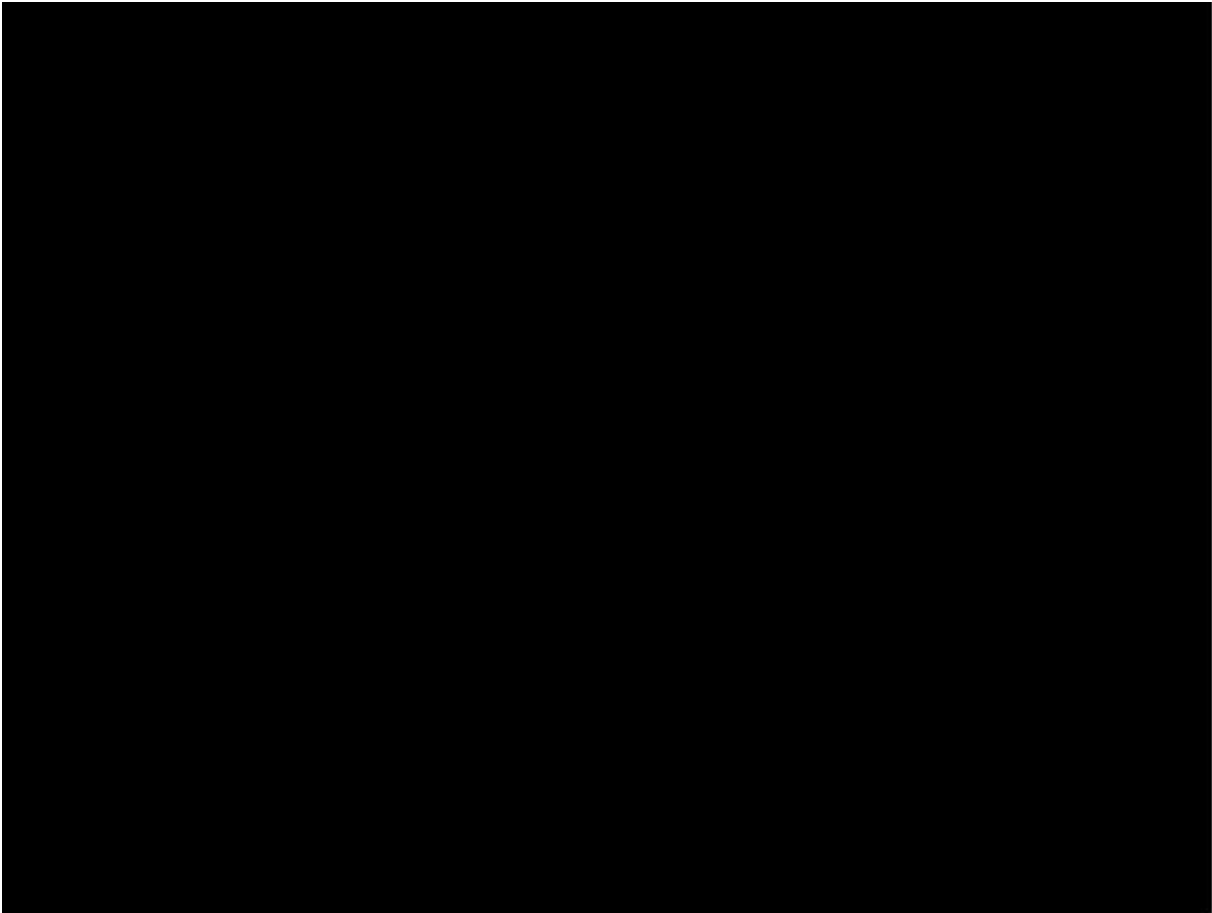
Explicitly, with member function `sync()`: Calling stream's member function `sync()`, which takes no parameters, causes an immediate synchronization. This function returns an `int` value equal to `-1` if the stream has no associated buffer or in case of failure. Otherwise (if the stream buffer was successfully synchronized) it returns `0`.

C++ Fundamentals



Table of Contents

CRITICAL SKILL 1.1: A Brief History of C++	2
--	---



CRITICAL SKILL 1.2: How C++ Relates to Java and C#	5
CRITICAL SKILL 1.3: Object-Oriented Programming	7
CRITICAL SKILL 1.4: A First Simple Program	10
CRITICAL SKILL 1.5: A Second Simple Program	15
CRITICAL SKILL 1.6: Using an Operator	17
CRITICAL SKILL 1.7: Reading Input from the Keyboard	19
Project 1-1 Converting Feet to Meters	24
CRITICAL SKILL 1.8: Two Control Statements	26
CRITICAL SKILL 1.9: Using Blocks of Code	30
Project 1-2 Generating a Table of Feet to Meter Conversions	33

CRITICAL SKILL 1.10: Introducing Functions 35

CRITICAL SKILL 1.11: The C++ Keywords 38

CRITICAL SKILL 1.12: Identifiers 39

If there is one language that defines the essence of programming today, it is C++. It is the preeminent language for the development of high-performance software. Its syntax has become the standard for professional programming languages, and its design philosophy reverberates throughout computing.

1	

C++ is also the language from which both Java and C# are derived. Simply stated, to be a professional programmer implies competency in C++. It is the gateway to all of modern programming.

The purpose of this module is to introduce C++, including its history, its design philosophy, and several of its most important features. By far, the hardest thing about learning a programming language is the fact that no element exists in isolation. Instead, the components of the language work together. This interrelatedness makes it difficult to discuss one aspect of C++ without involving others. To help overcome this problem, this module provides a brief overview of several C++ features, including the general form of a C++ program, some basic control statements, and operators. It does not go into too many details, but rather concentrates on the general concepts common to any C++ program.

CRITICAL SKILL 1.1: A Brief History of C++

The history of C++ begins with C. The reason for this is easy to understand: C++ is built upon the foundation of C. Thus, C++ is a superset of C. C++ expanded and enhanced the C language to support object-oriented programming (which is described later in this module). C++ also added several other improvements to the C language, including an extended set of library routines. However, much of the spirit and flavor of C++ is directly inherited from C. Therefore, to fully understand and appreciate C++, you need to understand the “how and why” behind C.

C: The Beginning of the Modern Age of Programming

The invention of C defines the beginning of the modern age of programming. Its impact should not be underestimated because it fundamentally changed the way programming was approached and thought about. Its design philosophy and syntax have influenced every major language since. C was one of the major, revolutionary forces in computing.

C was invented and first implemented by Dennis Ritchie on a DEC PDP-11 using the UNIX operating system. C is the result of a development process

that started with an older language called BCPL. BCPL was developed by Martin Richards. BCPL influenced a language called B, which was invented by Ken Thompson and which led to the development of C in the 1970s.

Prior to the invention of C, computer languages were generally designed either as academic exercises or by bureaucratic committees. C was different. It was designed, implemented, and developed by real, working programmers, reflecting the way they approached the job of programming. Its features were honed, tested, thought about, and rethought by the people who actually used the language. As a result, C attracted many proponents and quickly became the language of choice of programmers around the world.

C grew out of the structured programming revolution of the 1960s. Prior to structured programming, large programs were difficult to write because the program logic tended to degenerate into what is known as “spaghetti code,” a tangled mass of jumps, calls, and returns that is difficult to follow. Structured languages addressed this problem by adding well-defined control statements, subroutines

2	

with local variables, and other improvements. Using structured languages, it became possible to write moderately large programs.

Although there were other structured languages at the time, such as Pascal, C was the first to successfully combine power, elegance, and expressiveness. Its terse, yet easy-to-use syntax coupled with its philosophy that the programmer (not the language) was in charge quickly won many converts. It can be a bit hard to understand from today's perspective, but C was a breath of fresh air that programmers had long awaited. As a result, C became the most widely used structured programming language of the 1980s.

The Need for C++

Given the preceding discussion, you might be wondering why C++ was invented. Since C was a successful computer programming language, why was there a need for something else? The answer is complexity. Throughout the history of programming, the increasing complexity of programs has driven the need for better ways to manage that complexity. C++ is a response to that need. To better understand the correlation between increasing program complexity and computer language development, consider the following.

Approaches to programming have changed dramatically since the invention of the computer. For example, when computers were first invented, programming was done by using the computer's front panel to toggle in the binary machine instructions. As long as programs were just a few hundred instructions long, this approach worked. As programs grew, assembly language was invented so that programmers could deal with larger, increasingly complex programs by using symbolic representations of the machine instructions. As programs continued to grow, high-level languages were developed to give programmers more tools with which to handle the complexity.

The first widely used computer language was, of course, FORTRAN. While FORTRAN was a very impressive first step, it is hardly a language that encourages clear, easy-to-understand programs. The 1960s gave birth to structured programming, which is the method of programming encouraged by languages such as C. With structured languages it was, for the first time, possible to write moderately complex programs fairly easily.

However, even with structured programming methods, once a project reaches a certain size, its complexity exceeds what a programmer can manage. By the late 1970s, many projects were near or at this point.

In response to this problem, a new way to program began to emerge: object-oriented programming (OOP). Using OOP, a programmer could handle larger, more complex programs. The trouble was that C did not support object-oriented programming. The desire for an object-oriented version of C ultimately led to the creation of C++.

In the final analysis, although C is one of the most liked and widely used professional programming languages in the world, there comes a time when its ability to handle complexity is exceeded. Once a program reaches a certain size, it becomes so complex that it is difficult to grasp as a totality. The

3	

purpose of C++ is to allow this barrier to be broken and to help the programmer comprehend and manage larger, more complex programs.

C++ Is Born

C++ was invented by Bjarne Stroustrup in 1979, at Bell Laboratories in Murray Hill, New Jersey. He initially called the new language “C with Classes.” However, in 1983 the name was changed to C++.

Stroustrup built C++ on the foundation of C, including all of C’s features, attributes, and benefits. He also adhered to C’s underlying philosophy that the programmer, not the language, is in charge. At this point, it is critical to understand that Stroustrup did not create an entirely new programming language.

Instead, he enhanced an already highly successful language.

Most of the features that Stroustrup added to C were designed to support object-oriented programming. In essence, C++ is the object-oriented version of C. By building upon the foundation of C, Stroustrup provided a smooth migration path to OOP. Instead of having to learn an entirely new language, a C programmer needed to learn only a few new features before reaping the benefits of the object-oriented methodology.

When creating C++, Stroustrup knew that it was important to maintain the original spirit of C, including its efficiency, flexibility, and philosophy, while at the same time adding support for object-oriented programming. Happily, his goal was accomplished. C++ still provides the programmer with the freedom and control of C, coupled with the power of objects.

Although C++ was initially designed to aid in the management of very large programs, it is in no way limited to this use. In fact, the object-oriented attributes of C++ can be effectively applied to virtually any programming task. It is not uncommon to see C++ used for projects such as editors, databases, personal file systems, networking utilities, and communication programs. Because C++ shares C’s efficiency, much high-performance systems software is constructed using C++. Also, C++ is frequently the language of choice for Windows programming.

The Evolution of C++

Since C++ was first invented, it has undergone three major revisions, with each revision adding to and altering the language. The first revision was in 1985 and the second in 1990. The third occurred during the C++ standardization process. Several years ago, work began on a standard for C++. Toward that end, a joint ANSI (American National Standards Institute) and ISO (International Standards Organization) standardization committee was formed. The first draft of the proposed standard was created on January 25, 1994. In that draft, the ANSI/ISO C++ committee (of which I was a member) kept the features first defined by Stroustrup and added some new ones. But, in general, this initial draft reflected the state of C++ at the time.

Soon after the completion of the first draft of the C++ standard, an event occurred that caused the standard to be greatly expanded: the creation of the Standard Template Library (STL) by Alexander Stepanov. The STL is a set of generic routines that you can use to manipulate data. It is both powerful

4	

and elegant. But it is also quite large. Subsequent to the first draft, the committee voted to include the STL in the specification for C++. The addition of the STL expanded the scope of C++ well beyond its original definition. While important, the inclusion of the STL, among other things, slowed the standardization of C++.

It is fair to say that the standardization of C++ took far longer than anyone had expected. In the process, many new features were added to the language, and many small changes were made. In fact, the version of C++ defined by the ANSI/ISO C++ committee is much larger and more complex than Stroustrup's original design. The final draft was passed out of committee on November 14, 1997, and an ANSI/ISO standard for C++ became a reality in 1998. This is the specification for C++ that is usually referred to as Standard C++.

The material in this book describes Standard C++. This is the version of C++ supported by all mainstream C++ compilers, including Microsoft's Visual C++. Thus, the code and information in this book are fully portable.

CRITICAL SKILL 1.2: How C++ Relates to Java and C#

In addition to C++, there are two other important, modern programming languages: Java and C#. Java was developed by Sun Microsystems, and C# was created by Microsoft. Because there is sometimes confusion about how these two languages relate to C++, a brief discussion of their relationship is in order.

C++ is the parent for both Java and C#. Although both Java and C# added, removed, and modified various features, in total the syntax for these three languages is nearly identical. Furthermore, the object model used by C++ is similar to the ones used by Java and C#. Finally, the overall “look and feel” of these languages is very similar. This means that once you know C++, you can easily learn Java or C#. The opposite is also true. If you know Java or C#, learning C++ is easy. This is one reason that Java and C# share C++'s syntax and object model; it facilitated their rapid adoption by legions of experienced C++ programmers.

The main difference between C++, Java, and C# is the type of computing

Q: How do Java and C# create cross-platform, portable programs, and why can't C++ do the same?

A: Java and C# can create cross-platform, portable programs and C++

can't because of the type of object code produced by the compiler. In the case of C++, the output from the compiler is machine code

5	

that is directly executed by the CPU. Thus, it is tied to a specific CPU and operating system. If you want to run a C++ program on a different system, you need to recompile it into machine code specifically targeted for that environment. To create a C++ program that would run in a variety of environments, several different executable versions of the program are needed.

Java and C# achieve portability by compiling a program into a pseudocode, intermediate language. In the case of Java, this intermediate language is called bytecode. For C#, it is called Microsoft Intermediate Language (MSIL). In both cases, this pseudocode is executed by a runtime system. For Java, this runtime system is called the Java Virtual Machine (JVM). For C#, it is the Common Language Runtime (CLR). Therefore, a Java program can run in any environment for which a JVM is available, and a C#

program can run in any environment in which the CLR is implemented.

Since the Java and C# runtime systems stand between a program and the CPU, Java and C# programs incur an overhead that is not present in the execution of a C++ program. This is why C++ programs usually run faster than the equivalent programs written in Java or C#.

Java and C# were developed in response to the unique programming needs of the online environment of the Internet. (C# was also designed to simplify the creation of software components.) The Internet is connected to many different types of CPUs and operating systems. Thus, the ability to produce cross-platform, portable programs became an overriding concern.

The first language to address this need was Java. Using Java, it is possible to write a program that runs in a wide variety of environments. Thus, a Java program can move about freely on the Internet. However, the price you pay for portability is efficiency, and Java programs execute more slowly than do C++ programs. The same is true for C#. In the final analysis, if you want to create high-performance software, use C++. If you need to create highly portable software, use Java or C#.

One final point: Remember that C++, Java, and C# are designed to solve different sets of problems. It is not an issue of which language is best in and of itself. Rather, it is a question of which language is right for the job at hand.



From what language is C++ derived?

What was the main factor that drove the creation of C++?

C++ is the parent of Java and C#. True or False?

6	

Answer Key:

C++ is derived from C.

Increasing program complexity was the main factor that drove the creation of C++.

True.

CRITICAL SKILL 1.3: Object-Oriented

Programming

Central to C++ is object-oriented programming (OOP). As just explained, OOP was the impetus for the creation of C++. Because of this, it is useful to understand OOP's basic principles before you write even a simple C++ program.

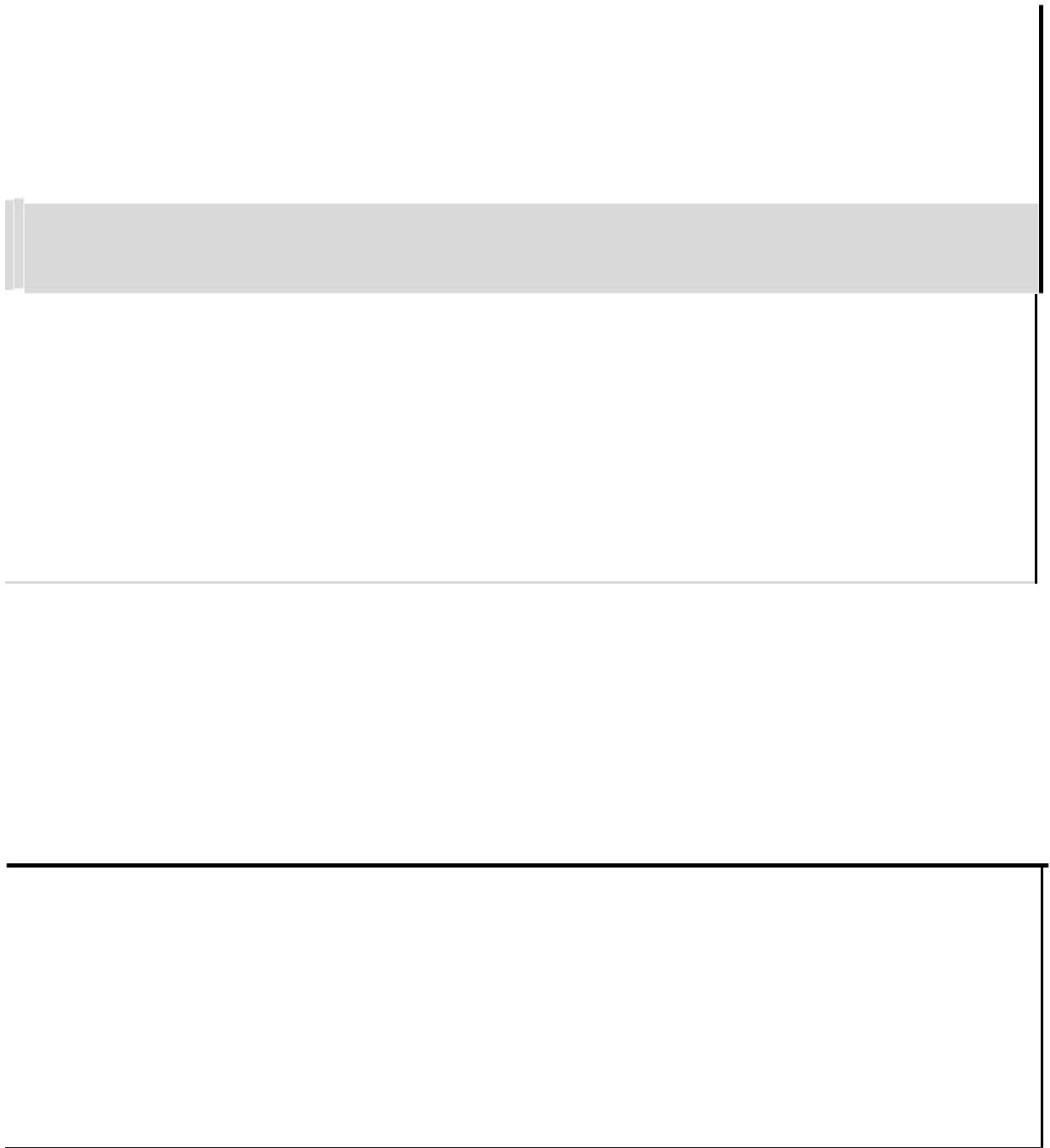
Object-oriented programming took the best ideas of structured programming and combined them with several new concepts. The result was a different and better way of organizing a program. In the most general sense, a program can be organized in one of two ways: around its code (what is happening) or around its data (who is being affected). Using only structured programming techniques, programs are typically organized around code. This approach can be thought of as “code acting on data.”

Object-oriented programs work the other way around. They are organized around data, with the key principle being “data controlling access to code.” In an object-oriented language, you define the data and the routines that are permitted to act on that data. Thus, a data type defines precisely what sort of operations can be applied to that data.

To support the principles of object-oriented programming, all OOP languages, including C++, have three traits in common: encapsulation, polymorphism, and inheritance. Let's examine each.

Encapsulation

Encapsulation is a programming mechanism that binds together code and the data it manipulates, and that keeps both safe from outside interference and misuse. In an object-oriented language, code and data can be bound together in such a way that a self-contained black box is created. Within the box are all necessary data and code. When code and data are linked together in this fashion, an object is created. In other words, an object is the device that supports encapsulation.



Ask the Expert

Q: I have heard the term method applied to a subroutine. Is a method the same as a function?

A: In general, the answer is yes. The term method was popularized by

Java. What a C++ programmer calls a function, a Java programmer calls a method. C# programmers also use the term method. Because it is becoming so widely used, sometimes the term method is also used when referring to a C++

7	

function.

Within an object, code or data or both may be private to that object or public. Private code or data is known to and accessible by only another part of the object. That is, private code or data cannot be accessed by a piece of the program that exists outside the object. When code or data is public, other parts of your program can access it even though it is defined within an object. Typically, the public parts of an object are used to provide a controlled interface to the private elements of the object.

C++'s basic unit of encapsulation is the class. A class defines the form of an object. It specifies both the data and the code that will operate on that data. C++ uses a class specification to construct objects. Objects are instances of a class. Thus, a class is essentially a set of plans that specifies how to build an object.

The code and data that constitute a class are called members of the class. Specifically, member variables, also called instance variables, are the data defined by the class. Member functions are the code that operates on that data. Function is C++'s term for a subroutine.

Polymorphism

Polymorphism (from Greek, meaning “many forms”) is the quality that allows one interface to access a general class of actions. A simple example of polymorphism is found in the steering wheel of an automobile. The steering wheel (the interface) is the same no matter what type of actual steering mechanism is used. That is, the steering wheel works the same whether your car has manual steering, power steering, or rack-and-pinion steering. Thus, turning the steering wheel left causes the car to go left no matter what type of steering is used. The benefit of the uniform interface is, of course, that once you know how to operate the steering wheel, you can drive any type of car.

The same principle can also apply to programming. For example, consider a stack (which is a first-in, last-out list). You might have a program that requires three different types of stacks. One stack is used for integer values, one for floating-point values, and one for characters. In this case, the algorithm that implements each stack is the same, even though the data being stored differs. In a non-object-oriented language, you would be required to create three different sets of stack routines, with each set using different names. However, because of polymorphism, in C++ you can create one general set of stack routines that works for all three situations. This way, once you know how to use one stack, you can use them all.

More generally, the concept of polymorphism is often expressed by the phrase “one interface, multiple methods.” This means that it is possible to design a generic interface to a group of related activities. Polymorphism helps reduce complexity by allowing the same interface to specify a general class of action. It is the compiler’s job to select the specific action (that is, method) as it applies to each situation. You, the programmer, don’t need to do this selection manually. You need only remember and utilize the general interface.

8	

Inheritance

Inheritance is the process by which one object can acquire the properties of another object. This is important because it supports the concept of hierarchical classification. If you think about it, most knowledge is made manageable by hierarchical (that is, top-down) classifications. For example, a Red Delicious apple is part of the classification apple, which in turn is part of the fruit class, which is under the larger class food. That is, the food class possesses certain qualities (edible, nutritious, and so on) which also, logically, apply to its subclass, fruit. In addition to these qualities, the fruit class has specific characteristics (juicy, sweet, and so on) that distinguish it from other food. The apple class defines those qualities specific to an apple (grows on trees, not tropical, and so on). A Red Delicious apple would, in turn, inherit all the qualities of all preceding classes and would define only those qualities that make it unique.

Without the use of hierarchies, each object would have to explicitly define all of its characteristics. Using inheritance, an object need only define those qualities that make it unique within its class. It can inherit its general attributes from its parent. Thus, it is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case.



Name the principles of OOP.

What is the basic unit of encapsulation in C++?

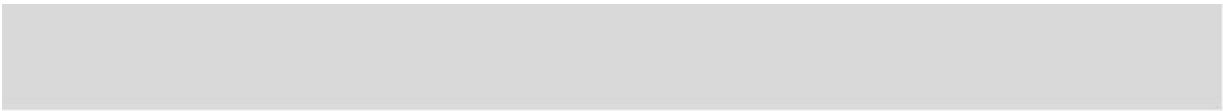
What is the commonly used term for a subroutine in C++?

Answer Key:

Encapsulation, polymorphism, and inheritance are the principles of OOP.

The class is the basic unit of encapsulation in C++.

Function is the commonly used term for a subroutine in C++.



--

Ask the Expert

Q: You state that object-oriented programming (OOP) is an effective way to manage large programs. However, it seems that OOP might add substantial overhead to relatively small ones. As it relates to C++,

9	

is this true?

A: No. A key point to understand about C++ is that it allows you to write object-oriented programs, but does not require you to do so. This is one of the important differences between C++ and Java/C#, which employ a strict object-model in which every program is, to at least a small extent, object oriented. C++ gives you the option. Furthermore, for the most part, the object-oriented features of C++ are transparent at runtime, so little (if any) overhead is incurred.

CRITICAL SKILL 1.4: A First Simple Program

Now it is time to begin programming. Let's start by compiling and running the short sample C++ program shown here:

```
/*
```

This is a simple C++ program.

Call this file Sample.cpp.

```

*/

#include <iostream>

using namespace std;

A C++ program begins at main(). int main()
{

    cout << "C++ is power programming.";

    return 0;

}

```

You will follow these three steps:

Enter the program.

Compile the program.

Run the program.

Before beginning, let's review two terms: source code and object code. Source code is the human-readable form of the program. It is stored in a text file. Object code is the executable form of the program created by the compiler.

10	

Entering the Program

The programs shown in this book are available from Osborne's web site: www.osborne.com. However, if you want to enter the programs by hand, you are free to do so. Typing in the programs yourself often helps you remember the key concepts. If you choose to enter a program by hand, you must use a text editor, not a word processor. Word processors typically store format information along with text. This format information will confuse the C++ compiler. If you are using a Windows platform, then you can use WordPad, or any other programming editor that you like.

The name of the file that holds the source code for the program is technically arbitrary. However, C++ programs are normally contained in files that use the file extension .cpp. Thus, you can call a C++ program file by any name, but it should use the .cpp extension. For this first example, name the source file Sample.cpp so that you can follow along. For most of the other programs in this book, simply use a name of your own choosing.

Compiling the Program

How you will compile Sample.cpp depends upon your compiler and what options you are using. Furthermore, many compilers, such as Microsoft's [Visual C++ Express Edition](#) which you can download for free, provide two different ways for compiling a program: the command-line compiler and the Integrated Development Environment (IDE). Thus, it is not possible to give generalized instructions for compiling a C++ program. You must consult your compiler's instructions.

The preceding paragraph notwithstanding, if you are using Visual C++, then the easiest way to compile and run the programs in this book is to use the command-line compilers offered by these environments. For example, to compile Sample.cpp using Visual C++, you will use this command line:

```
C:\...>cl -GX Sample.cpp
```

The -GX option enhances compilation. To use the Visual C++ command-line compiler, you must first execute the batch file VCVARS32.BAT, which is provided by Visual C++. (Visual Studio also provides a ready-to-use command prompt environment that can be activated by selecting Visual Studio Command Prompt from the list of tools shown under the Microsoft

Visual Studio entry in the Start | Programs menu of the taskbar.)

The output from a C++ compiler is executable object code. For a Windows environment, the executable file will use the same name as the source file, but have the .exe extension. Thus, the executable version of Sample.cpp will be in Sample.exe.

Run the Program

After a C++ program has been compiled, it is ready to be run. Since the output from a C++ compiler is executable object code, to run the program, simply enter its name at the command prompt. For example, to run Sample.exe, use this command line:

C:\...	>Sample
11	

When run, the program displays the following output:

C++ is power programming.

If you are using an Integrated Development Environment, then you can run a program by selecting Run from a menu. Consult the instructions for your specific compiler. For the programs in this book, it is usually easier to compile and run from the command line.

One last point: The programs in this book are console based, not window based. That is, they run in a Command Prompt session. C++ is completely at home with Windows programming. Indeed, it is the most commonly used language for Windows development. However, none of the programs in this book use the Windows Graphic User Interface (GUI). The reason for this is easy to understand: Windows programs are, by their nature, large and complex. The overhead required to create even a minimal Windows skeletal program is 50 to 70 lines of code. To write Windows programs that demonstrate the features of C++ would require hundreds of lines of code each. In contrast, console-based programs are much shorter and are the type of programs normally used to teach programming. Once you have mastered C++, you will be able to apply your knowledge to Windows programming with no trouble.

The First Sample Program Line by Line

Although Sample.cpp is quite short, it includes several key features that are common to all C++ programs. Let's closely examine each part of the program. The program begins with the lines

```
/*
```

This is a simple C++ program.

Call this file Sample.cpp.

```
*/
```

This is a comment. Like most other programming languages, C++ lets you enter a remark into a program's source code. The contents of a comment are ignored by the compiler. The purpose of a comment is to describe or explain the operation of a program to anyone reading its source code. In the

case of this comment, it identifies the program. In more complex programs, you will use comments to help explain what each feature of the program is for and how it goes about doing its work. In other words, you can use comments to provide a “play-by-play” description of what your program does.

In C++, there are two types of comments. The one you’ve just seen is called a multiline comment. This type of comment begins with a `/*` (a slash followed by an asterisk). It ends only when a `*/` is encountered. Anything between these two comment symbols is completely ignored by the compiler. Multiline comments may be one or more lines long. The second type of comment (single-line) is found a little further on in the program and will be discussed shortly.

The next line of code looks like this:

```
#include <iostream>
```

12	

The C++ language defines several headers, which contain information that is either necessary or useful to your program. This program requires the header `iostream`, which supports the C++ I/O system. This header is provided with your compiler. A header is included in your program using the `#include` directive. Later in this book, you will learn more about headers and why they are important.

The next line in the program is

```
using namespace std;
```

This tells the compiler to use the `std` namespace. Namespaces are a relatively recent addition to C++. Although namespaces are discussed in detail later in this book, here is a brief description. A namespace creates a declarative region in which various program elements can be placed. Elements declared in one namespace are separate from elements declared in another. Namespaces help in the organization of large programs. The `using` statement informs the compiler that you want to use the `std` namespace. This is the namespace in which the entire Standard C++ library is declared. By using the `std` namespace, you simplify access to the standard library. (Since namespaces are relatively new, an older compiler may not support them. If you are using an older compiler, see Appendix B, which describes an easy work-around.)

The next line in the program is

```
// A C++ program begins at main().
```

This line shows you the second type of comment available in C++: the single-line comment. Single-line comments begin with `//` and stop at the end of the line. Typically, C++ programmers use multiline comments when writing larger, more detailed commentaries, and single-line comments when short remarks are needed. This is, of course, a matter of personal style.

The next line, as the preceding comment indicates, is where program execution begins. `int main()`

All C++ programs are composed of one or more functions. As explained earlier, a function is a subroutine. Every C++ function must have a name, and the only function that any C++ program must include is the one shown here, called `main()`. The `main()` function is where program execution

begins and (most commonly) ends. (Technically speaking, a C++ program begins with a call to `main()` and, in most cases, ends when `main()` returns.) The opening curly brace on the line that follows `main()` marks the start of the `main()` function code. The `int` that precedes `main()` specifies the type of data returned by `main()`. As you will learn, C++ supports several built-in data types, and `int` is one of them. It stands for integer.

The next line in the program is

```
cout << "C++ is power programming.";
```

13	

This is a console output statement. It causes the message C++ is power programming. to be displayed on the screen. It accomplishes this by using the output operator <<. The << operator causes whatever expression is on its right side to be output to the device specified on its left side. cout is a predefined identifier that stands for console output and generally refers to the computer's screen. Thus, this statement causes the message to be output to the screen. Notice that this statement ends with a semicolon. In fact, all C++ statements end with a semicolon.

The message "C++ is power programming." is a string. In C++, a string is a sequence of characters enclosed between double quotes. Strings are used frequently in C++.

The next line in the program is

```
return 0;
```

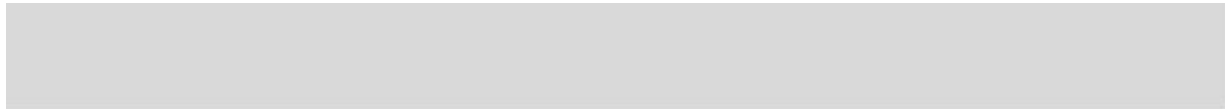
This line terminates main() and causes it to return the value 0 to the calling process (which is typically the operating system). For most operating systems, a return value of 0 signifies that the program is terminating normally. Other values indicate that the program is terminating because of some error. return is one of C++'s keywords, and it is used to return a value from a function. All of your programs should return 0 when they terminate normally (that is, without error).

The closing curly brace at the end of the program formally concludes the program.

Handling Syntax Errors

If you have not yet done so, enter, compile, and run the preceding program. As you may know from previous programming experience, it is quite easy to accidentally type something incorrectly when entering code into your computer. Fortunately, if you enter something incorrectly into your program, the compiler will report a syntax error message when it tries to compile it. Most C++ compilers attempt to make sense out of your source code no matter what you have written. For this reason, the error that is reported may not always reflect the actual cause of the problem. In the preceding program, for example, an accidental omission of the opening curly brace after main() may cause the compiler to report the cout

statement as the source of a syntax error. When you receive syntax error messages, be prepared to look at the last few lines of code in your program in order to find the error.



Ask the Expert

Q: In addition to error messages, my compiler offers several types of warning messages. How do warnings differ from errors, and what type of reporting should I use?

A: In addition to reporting fatal syntax errors, most C++ compilers can also report several types of warning messages. Error messages report things that are unequivocally wrong in your program, such as forgetting a semicolon. Warnings point out suspicious but technically correct code. You, the

14	

programmer, then decide whether the suspicion is justified.

Warnings are also used to report such things as inefficient constructs or the use of obsolete features. Generally, you can select the specific type of warnings that you want to see. The programs in this book are in compliance with Standard C++, and when entered correctly, they will not generate any troublesome warning messages.

For the examples in this book, you will want to use your compiler's default (or "normal") error reporting. However, you should examine your compiler's documentation to see what options you have at your disposal. Many compilers have sophisticated features that can help you spot subtle errors before they become big problems. Understanding your compiler's error reporting system is worth your time and effort.



Where does a C++ program begin execution?

What is cout?

What does `#include <iostream>` do?

Answer Key:

A C++ program begins execution with `main()`.

`cout` is a predefined identifier that is linked to console output.

It includes the header `<iostream>`, which supports I/O.

CRITICAL SKILL 1.5: A Second Simple Program

Perhaps no other construct is as fundamental to programming as the variable. A variable is a named memory location that can be assigned a value. Further, the value of a variable can be changed during the execution of a program. That is, the content of a variable is changeable, not fixed.

The following program creates a variable called `length`, gives it the value 7, and then displays the message “The length is 7” on the screen.

15	

```

// Using a variable.

#include <iostream>
using namespace std;

int main()
{
    int length; // this declares a variable ← Declare a variable.

    length = 7; // this assigns 7 to length ← Assign length a value.

    cout << "The length is ";
    cout << length; // This displays 7 ← Output the value in length.

    return 0;
}

```

As mentioned earlier, the names of C++ programs are arbitrary. Thus, when you enter this program, select a filename to your liking. For example, you could give this program the name VarDemo.cpp.

This program introduces two new concepts. First, the statement `int length; // this declares a variable`

declares a variable called `length` of type integer. In C++, all variables must be declared before they are used. Further, the type of values that the

variable can hold must also be specified. This is called the type of the variable. In this case, length may hold integer values. These are whole number values whose range will be at least $-32,768$ through $32,767$. In C++, to declare a variable to be of type integer, precede its name with the keyword `int`. Later, you will see that C++ supports a wide variety of built-in variable types. (You can create your own data types, too.)

The second new feature is found in the next line of code:

```
length = 7; // this assigns 7 to length
```

As the comment suggests, this assigns the value 7 to length. In C++, the assignment operator is the single equal sign. It copies the value on its right side into the variable on its left. After the assignment, the variable length will contain the number 7.

The following statement displays the value of length:

```
cout << length; // This displays 7
```

In general, if you want to display the value of a variable, simply put it on the right side of `<<` in a `cout` statement. In this specific case, because length contains the number 7, it is this number that is displayed

16	

on the screen. Before moving on, you might want to try giving length other values and watching the results.

CRITICAL SKILL 1.6: Using an Operator

Like most other computer languages, C+ supports a full range of arithmetic operators that enable you to manipulate numeric values used in a program. They include those shown here:

+	Addition
-	Subtraction
*	Multiplication
/	Division

These operators work in C++ just like they do in algebra.

The following program uses the * operator to compute the area of a rectangle given its length and the width.

```
// Using an operator.

#include <iostream>
using namespace std;

int main()
{
    int length; // this declares a variable
    int width;  // this declares another variable
    int area;   // this does, too

    length = 7; // this assigns 7 to length
    width = 5;  // this assigns 5 to width

    area = length * width; // compute area ← Assign the product of length
                                and width to area.

    cout << "The area is ";
    cout << area; // This displays 35

    return 0;
}
```

This program declares three variables: length, width, and area. It assigns the value 7 to length and the value 5 to width. It then computes the product and assigns that value to area. The program outputs the following:

The area is 35

17	

In this program, there is actually no need for the variable area. For example, the program can be rewritten like this:

```
// A simplified version of the area program.

#include <iostream>
using namespace std;

int main()
{
    int length; // this declares a variable
    int width;  // this declares another variable

    length = 7; // this assigns 7 to length

    width = 5;  // this assigns 5 to width

    cout << "The area is ";
    cout << length * width; // This displays 35 ←—— Output length * width directly.

    return 0;
}
```

In this version, the area is computed in the cout statement by multiplying

length by width. The result is then output to the screen.

One more point before we move on: It is possible to declare two or more variables using the same declaration statement. Just separate their names by commas. For example, length, width, and area could have been declared like this:

```
int length, width, area; // all declared using one statement
```

Declaring two or more variables in a single statement is very common in professionally written C++ code.



Must a variable be declared before it is used?

Show how to assign the variable min the value 0.

Can more than one variable be declared in a single declaration statement?



Answer Key:

Yes, in C++ variables must be declared before they are used.

```
min = 0;
```

Yes, two or more variables can be declared in a single declaration statement.

CRITICAL SKILL 1.7: Reading Input from the Keyboard

The preceding examples have operated on data explicitly specified in the program. For example, the area program just shown computes the area of a rectangle that is 7 by 5, and these dimensions are part of the program itself. Of course, the calculation of a rectangle's area is the same no matter what its size, so the program would be much more useful if it would prompt the user for the dimensions of the rectangle, allowing the user to enter them using the keyboard.

To enable the user to enter data into a program from the keyboard, you will use the `>>` operator. This is the C++ input operator. To read from the keyboard, use this general form

```
cin >> var;
```

Here, `cin` is another predefined identifier. It stands for console input and is automatically supplied by C++. By default, `cin` is linked to the keyboard, although it can be redirected to other devices. The variable that receives input is specified by `var`.

Here is the area program rewritten to allow the user to enter the dimensions of the rectangle:

19	

```
/*  
    An interactive program that computes  
    the area of a rectangle.  
*/  
  
#include <iostream>  
using namespace std;  
  
int main()  
{  
    int length; // this declares a variable  
    int width;  // this declares another variable  
  
    cout << "Enter the length: ";  
    cin >> length; // input the length ← Input the value of length  
                                                from the keyboard.  
  
    cout << "Enter the width: ";  
    cin >> width;  // input the width ← Input the value of width  
                                                from the keyboard.  
  
    cout << "The area is ";  
    cout << length * width; // display the area  
  
    return 0;  
}
```

Here is a sample run:

Enter the length: 8

Enter the width: 3

The area is 24

Pay special attention to these lines:

```
cout << "Enter the length: ";
```

```
cin >> length; // input the length
```

The `cout` statement prompts the user. The `cin` statement reads the user's response, storing the value in `length`. Thus, the value entered by the user (which must be an integer in this case) is put into the variable that is on the right side of the `>>` (in this case, `length`). Thus, after the `cin` statement executes, `length` will contain the rectangle's length. (If the user enters a nonnumeric response, `length` will be zero.) The statements that prompt and read the width work in the same way.

Some Output Options

So far, we have been using the simplest types of `cout` statements. However, `cout` allows much more sophisticated output statements. Here are two useful techniques. First, you can output more than one

20	

piece of information using a single cout statement. For example, in the area program, these two lines are used to display the area:

```
cout << "The area is ";  
  
cout << length * width;
```

These two statements can be more conveniently coded, as shown here:

```
cout << "The area is " << length * width;
```

This approach uses two output operators within the same cout statement. Specifically, it outputs the string “The area is” followed by the area. In general, you can chain together as many output operations as you like within one output statement. Just use a separate << for each item.

Second, up to this point, there has been no occasion to advance output to the next line— that is, to execute a carriage return–linefeed sequence. However, the need for this will arise very soon. In C++, the carriage return–linefeed sequence is generated using the newline character. To put a newline character into a string, use this code: \n (a backslash followed by a lowercase n). To see the effect of the \n, try the following program:

```
/*
```

This program demonstrates the \n code,
which generates a new line.

```
*/
```

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    cout << "one\n";
```

```
    cout << "two\n";
```

```
    cout << "three";
```

```
        cout << "four";  
  
        return 0;  
  
    }
```

This program produces the following output:

one

two

threefour

The newline character can be placed anywhere in the string, not just at the end. You might want to try experimenting with the newline character now, just to make sure you understand exactly what it does.

21	



Progress Check

What is C++'s input operator?

To what device is cin linked by default?

What does `\n` stand for?

Answer Key:

The input operator is >>.

cin is linked to the keyboard by default.

The \n stands for the newline character.

Another Data Type

In the preceding programs, variables of type int were used. However, a variable of type int can hold only whole numbers. Thus, it cannot be used when a fractional component is required. For example, an int variable can hold the value 18, but not the value 18.3. Fortunately, int is only one of several data types defined by C++. To allow numbers with fractional components, C++ defines two main flavors of floating-point types: float and double, which represent single- and double-precision values, respectively. Of the two, double is probably the most commonly used.

To declare a variable of type double, use a statement similar to that shown here:

```
double result;
```

Here, result is the name of the variable, which is of type double. Because result has a floating-point type, it can hold values such as 88.56, 0.034, or –107.03.

To better understand the difference between int and double, try the following program:

22	

```

/*
    This program illustrates the differences
    between int and double.
*/

#include <iostream>
using namespace std;

int main() {
    int ivar;    // this declares an int variable
    double dvar; // this declares a floating-point variable

    ivar = 100; // assign ivar the value 100

    dvar = 100.0; // assign dvar the value 100.0

    cout << "Original value of ivar: " << ivar << "\n";
    cout << "Original value of dvar: " << dvar << "\n";

    cout << "\n"; // print a blank line ←————— Output a blank line.

    // now, divide both by 3
    ivar = ivar / 3;
    dvar = dvar / 3.0;

    cout << "ivar after division: " << ivar << "\n";
    cout << "dvar after division: " << dvar << "\n";

    return 0;
}

```

The output from this program is shown here:

Original value of ivar: 100

Original value of dvar: 100

ivar after division: 33

dvar after division: 33.3333



--

23	

A: C++ supplies different data types so that you can write efficient programs. For example, integer arithmetic is faster than floating-point calculations. Thus, if you don't need fractional values, then you don't need to incur the overhead associated with types float or double. Also, the amount of memory required for one type of data might be less than that required for another. By supplying different types, C++ enables you to make the best use of system resources. Finally, some algorithms require (or at least benefit from) the use of a specific type of data. C++ supplies a number of built-in types to give you the greatest flexibility.

As you can see, when `ivar` is divided by 3, a whole-number division is performed and the outcome is 33—the fractional component is lost. However, when `dvar` is divided by 3, the fractional component is preserved.

There is one other new thing in the program. Notice this line:

```
cout << "\n"; // print a blank line
```

It outputs a newline. Use this statement whenever you want to add a blank line to your output.

Project 1-1 Converting Feet to Meters

Although the preceding sample programs illustrate several important features of the C++ language, they are not very useful. You may not know much about C++ at this point, but you can still put what you have learned to work to create a practical program. In this project, we will create a program that converts feet to meters. The program prompts the user for the number of feet. It then displays that value converted into meters.

A meter is equal to approximately 3.28 feet. Thus, we need to use floating-point data. To perform the conversion, the program declares two double variables. One will hold the number of feet, and the second will hold the conversion to meters.

Step by Step

Create a new C++ file called FtoM.cpp. (Remember, in C++ the name of the file is arbitrary, so you can use another name if you like.)

Begin the program with these lines, which explain what the program does, include the iostream header, and specify the std namespace.

```
/*
```

Project 1-1

This program converts feet to meters.
Call this program FtoM.cpp.

```
*/
```

24	


```
#include <iostream>
```

```
using namespace std;
```

Begin main() by declaring the variables f and m:

```
int main()
```

```
{
```

```
double f; // holds the length in feet double m; // holds  
the conversion to meters
```

Add the code that inputs the number of feet:

```
cout << "Enter the length in feet: "; cin >> f; // read the  
number of feet
```

Add the code that performs the conversion and displays the result:

```
m = f / 3.28; // convert to meters
```

```
cout << f << " feet is " << m << " meters.";
```

Conclude the program, as shown here: return 0; }

Your finished program should look like this:

```
/*
```

Project 1-1

This program converts feet to meters.
Call this program FtoM.cpp.

```
*/
```

```
#include <iostream> using namespace std;
```

```
int main()
```

```
{
```

```
    double f; // holds the length in feet  
    double m; // holds the conversion to  
    meters
```

```
    cout << "Enter the length in feet: "; cin  
    >> f; // read the number of feet
```

```
    m = f / 3.28; // convert to meters
```

```
    cout << f << " feet is " << m << "  
    meters.";
```

25	

```
        return 0;  
    }
```

Compile and run the program. Here is a sample run:


Enter the length in feet: 5 5 feet is 1.52439 meters.

Try entering other values. Also, try changing the program so that it converts meters to feet.

What is C++'s keyword for the integer data type?

What is double?

How do you output a newline?

 Progress Check

Answer Key:

The integer data type is `int`.

`double` is the keyword for the double floating-point data type.

To output a newline, use `\n`.

CRITICAL SKILL 1.8: Two Control Statements

Inside a function, execution proceeds from one statement to the next, top to bottom. It is possible, however, to alter this flow through the use of the various program control statements supported by C++. Although we will look closely at control statements later, two are briefly introduced here because we will be using them to write sample programs.

The if Statement

You can selectively execute part of a program through the use of C++'s conditional statement: the `if`. The `if` statement works in C++ much like the `IF` statement in any other language. For example, it is syntactically identical to the `if` statements in C, Java, and C#. Its simplest form is shown here:

`if(condition) statement;`

where `condition` is an expression that is evaluated to be either true or false. In C++, true is nonzero and false is zero. If the condition is true, then the statement will execute. If it is false, then the statement will

26	

not execute. For example, the following fragment displays the phrase 10 is less than 11 on the screen because 10 is less than 11.

```
if(10 < 11) cout << "10 is less than 11";
```

However, consider the following:

```
if(10 > 11) cout << "this does not display";
```

In this case, 10 is not greater than 11, so the cout statement is not executed. Of course, the operands inside an if statement need not be constants. They can also be variables.

C++ defines a full complement of relational operators that can be used in a conditional expression. They are shown here:

Operator	Meaning
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
==	Equal to
!=	Not equal

Notice that the test for equality is the double equal sign. Here is a program that illustrates the if statement:

```
// Demonstrate the if.
```

```
#include <iostream>
```

```
using namespace std;
```

27	

```

int main() {
    int a, b, c;

    a = 2;
    b = 3;

    if(a < b) cout << "a is less than b\n"; ← An if statement

    // this won't display anything
    if(a == b) cout << "you won't see this\n";

    cout << "\n";

    c = a - b; // c contains -1

    cout << "c contains -1\n";
    if(c >= 0) cout << "c is non-negative\n";
    if(c < 0) cout << "c is negative\n";

    cout << "\n";

    c = b - a; // c now contains 1
    cout << "c contains 1\n";
    if(c >= 0) cout << "c is non-negative\n";
    if(c < 0) cout << "c is negative\n";

    return 0;
}

```


The output generated by this program is shown here:

a is less than b

c contains -1

c is negative

c contains 1

c is non-negative

The for Loop

You can repeatedly execute a sequence of code by creating a loop. C++ supplies a powerful assortment of loop constructs. The one we will look at here is the for loop. If you are familiar with C# or Java, then you will be pleased to know that the for loop in C++ works the same way it does in those languages. The simplest form of the for loop is shown here:

for(initialization; condition; increment) statement;

Here, initialization sets a loop control variable to an initial value. condition is an expression that is tested each time the loop repeats. As long as condition is true (nonzero), the loop keeps running. The

28	

increment is an expression that determines how the loop control variable is incremented each time the loop repeats.

The following program demonstrates the for. It prints the numbers 1 through 100 on the screen.

```
// A program that illustrates the for loop.


#include <iostream>
using namespace std;

int main()
{
    int count;

    for(count=1; count <= 100; count=count+1)
        cout << count << " ";

    return 0;
}
```

This is a for loop.



In the loop, count is initialized to 1. Each time the loop repeats, the condition `count <= 100`

is tested. If it is true, the value is output and count is increased by one. When count reaches a value greater than 100, the condition becomes false, and the loop stops running. In professionally written C++ code, you will

almost never see a statement like

```
count=count+1
```

because C++ includes a special increment operator that performs this operation more efficiently. The increment operator is ++ (two consecutive plus signs). The ++ operator increases its operand by 1. For example, the preceding for statement will generally be written like this:

```
for(count=1; count <= 100; count++) cout << count << " ";
```

This is the form that will be used throughout the rest of this book.

C++ also provides a decrement operator, which is specified as --. It decreases its operand by 1.



1. What does the **if** statement do?

29	

What does the **for** statement do?

What are C++'s relational operators?

Answer Key:

if is C++'s conditional statement.

The for is one of C++'s loop statements.

The relational operators are ==, !=, <, >, <=, and >=.

CRITICAL SKILL 1.9: Using Blocks of Code

Another key element of C++ is the code block. A code block is a grouping of two or more statements. This is done by enclosing the statements between opening and closing curly braces. Once a block of code has been created, it becomes a logical unit that can be used any place that a single statement can. For example, a block can be a target of the if and for statements. Consider this if statement:

```
if(w < h) {  
  
    v = w * h;  
  
    w = 0;  
  
}
```

Here, if w is less than h , then both statements inside the block will be executed. Thus, the two statements inside the block form a logical unit, and one statement cannot execute without the other also executing. The key point here is that whenever you need to logically link two or more statements, you do so by creating a block. Code blocks allow many algorithms to be implemented with greater clarity and efficiency.

Here is a program that uses a block of code to prevent a division by zero:



30	

```
// Demonstrate a block of code.
```

```
#include <iostream>
using namespace std;
```

```
int main() {
    double result, n, d;
```

```
    cout << "Enter value: ";
    cin >> n;
```

```
    cout << "Enter divisor: ";
    cin >> d;
```

The target of this if
is the entire block.

```
    // the target of this if is a block
```

```
    if(d != 0) {
        cout << "d does not equal zero so division is OK" << "\n";
        result = n / d;
        cout << n << " / " << d << " is " << result;
    }
```

```
    return 0;
```

```
}
```

Here is a sample run:

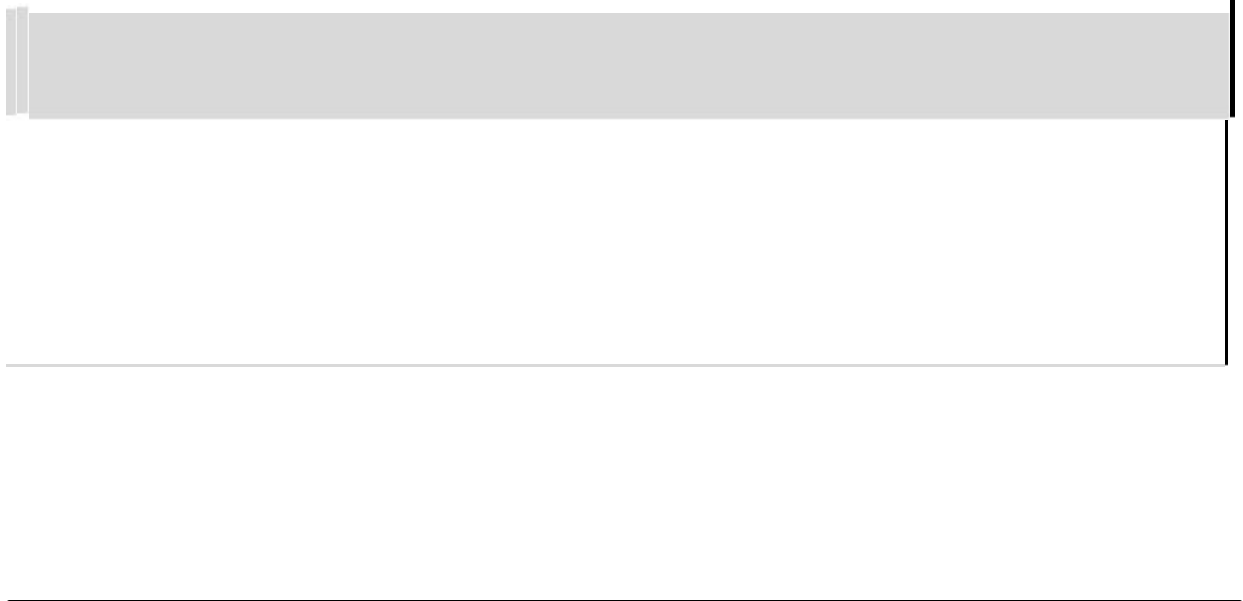
Enter value: 10

Enter divisor: 2

d does not equal zero so division is OK 10 / 2 is 5

In this case, the target of the if statement is a block of code and not just a single statement. If the condition controlling the if is true (as it is in the sample run), the three statements inside the block will be executed. Try entering a zero for the divisor and observe the result. In this case, the code inside the block is bypassed.

As you will see later in this book, blocks of code have additional properties and uses. However, the main reason for their existence is to create logically inseparable units of code.



Ask the Expert

Q: Does the use of a code block introduce any runtime inefficiencies? In other words, do the { and } consume any extra time during the execution of my program?

31	

A: No. Code blocks do not add any overhead whatsoever. In fact, because of their ability to simplify the coding of certain algorithms, their use generally increases speed and efficiency.

Semicolons and Positioning

In C++, the semicolon signals the end of a statement. That is, each individual statement must end with a semicolon. As you know, a block is a set of logically connected statements that is surrounded by opening and closing braces. A block is not terminated with a semicolon. Since a block is a group of statements, with a semicolon after each statement, it makes sense that a block is not terminated by a semicolon; instead, the end of the block is indicated by the closing brace.

C++ does not recognize the end of the line as the end of a statement—only a semicolon terminates a statement. For this reason, it does not matter where on a line you put a statement.

For example, to C++

```
x = y;
```

```
y = y + 1;
```

```
cout << x << " " << y;
```

is the same as

```
x = y; y = y + 1; cout << x << " " << y;
```

Furthermore, the individual elements of a statement can also be put on separate lines. For example, the following is perfectly acceptable:

```
cout << "This is a long line. The sum is : " << a + b + c +  
                                         d + e + f;
```

Breaking long lines in this fashion is often used to make programs more readable. It can also help prevent excessively long lines from wrapping.

Indentation Practices

You may have noticed in the previous examples that certain statements were indented. C++ is a free-form language, meaning that it does not matter where you place statements relative to each other on a line. However, over the years, a common and accepted indentation style has developed that allows for very readable programs. This book follows that style, and it is recommended that you do so as well. Using this style, you indent one level after each opening brace and move back out one level after each closing brace. There are certain statements that encourage some additional indenting; these will be covered later.

32	



Progress Check

How is a block of code created? What does it do?

In C++, statements are terminated by a

_____.

All C++ statements must start and end on one line. True or false?

Answer Key:

A block is started by a {. It is ended by a }. A block creates a logical unit of code.

semicolon

False.

Project 1-2 Generating a Table of Feet to Meter Conversions

This project demonstrates the for loop, the if statement, and code blocks to create a program that displays a table of feet-to-meters conversions. The table begins with 1 foot and ends at 100 feet. After every 10 feet, a blank line is output. This is accomplished through the use of a variable called counter that counts the number of lines that have been output. Pay special attention to its use.

Step by Step

Create a new file called FtoMTable.cpp.

Enter the following program into the file:

33	

```

/*
    Project 1-2

    This program displays a conversion table of feet to meters.

    Call this program FtoMTable.cpp.
*/

#include <iostream>
using namespace std;

int main() {
    double f; // holds the length in feet
    double m; // holds the conversion to meters
    int counter;

    counter = 0; ← Line counter is initially set to zero.

    for(f = 1.0; f <= 100.0; f++) {
        m = f / 3.28; // convert to meters
        cout << f << " feet is " << m << " meters.\n";

        counter++; ← Increment the line counter with each loop iteration.

        // every 10th line, print a blank line
        if(counter == 10) { ← If counter is 10,
            cout << "\n"; // output a blank line    output a blank line.
            counter = 0; // reset the line counter
        }
    }

    return 0;
}

```

Notice how counter is used to output a blank line after each ten lines. It is initially set to zero outside the for loop. Inside the loop, it is incremented after each conversion. When counter equals 10, a blank line is output, counter is reset to zero, and the process repeats.

Compile and run the program. Here is a portion of the output that you will see. Notice that results that don't produce an even result include a fractional component.

1 feet is 0.304878 meters.

2 feet is 0.609756 meters.

3 feet is 0.914634 meters.

4 feet is 1.21951 meters.

5 feet is 1.52439 meters.

6 feet is 1.82927 meters.

7 feet is 2.13415 meters.

34	

8 feet is 2.43902 meters.

9 feet is 2.7439 meters.

10 feet is 3.04878 meters.

11 feet is 3.35366 meters.

12 feet is 3.65854 meters.

13 feet is 3.96341 meters.

14 feet is 4.26829 meters.

15 feet is 4.57317 meters.

16 feet is 4.87805 meters.

17 feet is 5.18293 meters.

18 feet is 5.4878 meters.

19 feet is 5.79268 meters.

20 feet is 6.09756 meters.

21 feet is 6.40244 meters.

22 feet is 6.70732 meters.

23 feet is 7.0122 meters.

24 feet is 7.31707 meters.

25 feet is 7.62195 meters.

26 feet is 7.92683 meters.

27 feet is 8.23171 meters.

28 feet is 8.53659 meters.

29 feet is 8.84146 meters.

30 feet is 9.14634 meters.

31 feet is 9.45122 meters.

32 feet is 9.7561 meters.

33 feet is 10.061 meters.

34 feet is 10.3659 meters.

35 feet is 10.6707 meters.

36 feet is 10.9756 meters.

37 feet is 11.2805 meters.

38 feet is 11.5854 meters.

39 feet is 11.8902 meters.

40 feet is 12.1951 meters.

5. On your own, try changing this program so that it prints a blank line every 25 lines.

CRITICAL SKILL 1.10: Introducing Functions

A C++ program is constructed from building blocks called functions. Although we will look at the function in detail in Module 5, a brief overview is useful now. Let's begin by defining the term function: a function is a subroutine that contains one or more C++ statements.

Each function has a name, and this name is used to call the function. To call a function, simply specify its name in the source code of your program, followed by parentheses. For example, assume some function named MyFunc. To call MyFunc, you would write

35	

```
MyFunc();
```

When a function is called, program control is transferred to that function, and the code contained within the function is executed. When the function's code ends, control is transferred back to the caller. Thus, a function performs a task for other parts of a program.

Some functions require one or more arguments, which you pass when the function is called. Thus, an argument is a value passed to a function. Arguments are specified between the opening and closing parentheses when a function is called. For example, if `MyFunc()` requires an integer argument, then the following calls `MyFunc()` with the value 2:

```
MyFunc(2);
```

When there are two or more arguments, they are separated by commas. In this book, the term argument list will refer to comma-separated arguments. Remember, not all functions require arguments. When no argument is needed, the parentheses are empty.

A function can return a value to the calling code. Not all functions return values, but many do. The value returned by a function can be assigned to a variable in the calling code by placing the call to the function on the right side of an assignment statement. For example, if `MyFunc()` returned a value, it could be called as shown here:

```
x = MyFunc(2);
```

This statement works as follows. First, `MyFunc()` is called. When it returns, its return value is assigned to `x`. You can also use a call to a function in an expression. For example,

```
x = MyFunc(2) + 10;
```

In this case, the return value from `MyFunc()` is added to 10, and the result is assigned to `x`. In general, whenever a function's name is encountered in a statement, it is automatically called so that its return value can be obtained.

To review: an argument is a value passed into a function. A return value is data that is passed back to the calling code.

Here is a short program that demonstrates how to call a function. It uses

one of C++'s built-in functions, called `abs()`, to display the absolute value of a number. The `abs()` function takes one argument, converts it into its absolute value, and returns the result.

36	

```

// Use the abs() function.

#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
    int result;

    result = abs(-10); ← The calls the abs( ) function and
                        assigns its return value to result.

    cout << result;

    return 0;
}

```

Here, the value -10 is passed as an argument to `abs()`. The `abs()` function receives the argument with which it is called and returns its absolute value, which is 10 in this case. This value is assigned to `result`. Thus, the program displays “10” on the screen.

Notice one other thing about the preceding program: it includes the header `cstdlib`. This is the header required by `abs()`. Whenever you use a built-in function, you must include its header.

In general, there are two types of functions that will be used by your programs. The first type is written by you, and `main()` is an example of this type of function. Later, you will learn how to write other functions of your own. As you will see, real-world C++ programs contain many user-written functions.

The second type of function is provided by the compiler. The `abs()` function used by the preceding program is an example. Programs that you write will generally contain a mix of functions that you create and those supplied by the compiler.

When denoting functions in text, this book has used and will continue to use a convention that has become common when writing about C++. A function will have parentheses after its name. For example, if a function's name is `getval`, then it will be written `getval()` when its name is used in a sentence. This notation will help you distinguish variable names from function names in this book.

The C++ Libraries

As just explained, `abs()` is provided with your C++ compiler. This function and many others are found in the standard library. We will be making use of library functions in the example programs throughout this book.

C++ defines a large set of functions that are contained in the standard function library. These functions perform many commonly needed tasks, including I/O operations, mathematical computations, and

37	

string handling. When you use a library function, the C++ compiler automatically links the object code for that function to the object code of your program.

Because the C++ standard library is so large, it already contains many of the functions that you will need to use in your programs. The library functions act as building blocks that you simply assemble. You should explore your compiler's library documentation. You may be surprised at how varied the library functions are. If you write a function that you will use again and again, it too can be stored in a library.

In addition to providing library functions, every C++ compiler also contains a class library, which is an object-oriented library. However, you will need to wait until you learn about classes and objects before you can make use of the class library.



What is a function?

A function is called by using its name.

True or false?

What is the C++ standard function library?

Answer Key:

A function is a subroutine that contains one or more C++ statements.

True.

The C++ standard function library is a collection of functions supplied by all C++ compilers.

CRITICAL SKILL 1.11: The C++ Keywords

There are 63 keywords currently defined for Standard C++. These are shown in Table 1-1. Together with the formal C++ syntax, they form the

C++ programming language. Also, early versions of C++ defined the overload keyword, but it is obsolete. Keep in mind that C++ is a case-sensitive language, and it requires that all keywords be in lowercase.

38	

asm	auto	bool	break
case	catch	char	class
const	const_cast	continue	default
delete	do	double	dynamic_cast
else	enum	explicit	export
extern	false	float	for
friend	goto	if	inline
int	long	mutable	namespace
new	operator	private	protected
public	register	reinterpret_cast	return
short	signed	sizeof	static
static_cast	struct	switch	template
this	throw	true	try
typedef	typeid	typename	union
unsigned	using	virtual	void
volatile	wchar_t	while	

Table 1-1 The C++ Keywords

CRITICAL SKILL 1.12: Identifiers

In C++, an identifier is a name assigned to a function, variable, or any other user-defined item. Identifiers can be from one to several characters long. Variable names can start with any letter of the alphabet or an underscore. Next comes a letter, a digit, or an underscore. The underscore can be used to enhance the readability of a variable name, as in `line_count`. Uppercase and lowercase are seen as different; that is, to C++, `myvar` and `MyVar` are separate names. There is one important identifier restriction: you cannot use any of the C++ keywords as identifier names. In addition, predefined identifiers such as `cout` are also off limits.

Here are some examples of valid identifiers:

Test	x	y2	MaxIncr
up	_top	my_var	simpleInterest23

Remember, you cannot start an identifier with a digit. Thus, `98OK` is invalid. Good programming practice dictates that you use identifier names that reflect the meaning or usage of the items being named.

39	



Progress Check

Which is the keyword, for, For, or FOR?

A C++ identifier can contain what type of characters?

Are `index21` and `Index21` the same identifier?

Answer Key:

The keyword is for. In C++, all keywords are in lowercase.

A C++ identifier can contain letters, digits, and the underscore.

No, C++ is case sensitive.



Module 1 Mastery Check

It has been said that C++ sits at the center of the modern programming universe. Explain this statement.

A C++ compiler produces object code that is directly executed by the computer. True or false?

What are the three main principles of object-oriented programming?

Where do C++ programs begin execution?

What is a header?

What is `<iostream>`? What does the following code do?

```
#include <iostream>
```

What is a namespace?

What is a variable?

Which of the following variable names is/are invalid?

40	

count

_count

count27

67count

if

How do you create a single-line comment? How do you create a multiline comment?

Show the general form of the if statement. Show the general form of the for loop.

How do you create a block of code?

The moon's gravity is about 17 percent that of Earth's. Write a program that displays a table that shows Earth pounds and their equivalent moon weight. Have the table run from 1 to 100 pounds. Output a newline every 25 pounds.

A year on Jupiter (the time it takes for Jupiter to make one full circuit around the Sun) takes about 12 Earth years. Write a program that converts Jovian years to Earth years. Have the user specify the number of Jovian years. Allow fractional years.

When a function is called, what happens to program control?

Write a program that averages the absolute value of five values entered by the user. Display the result.

41	

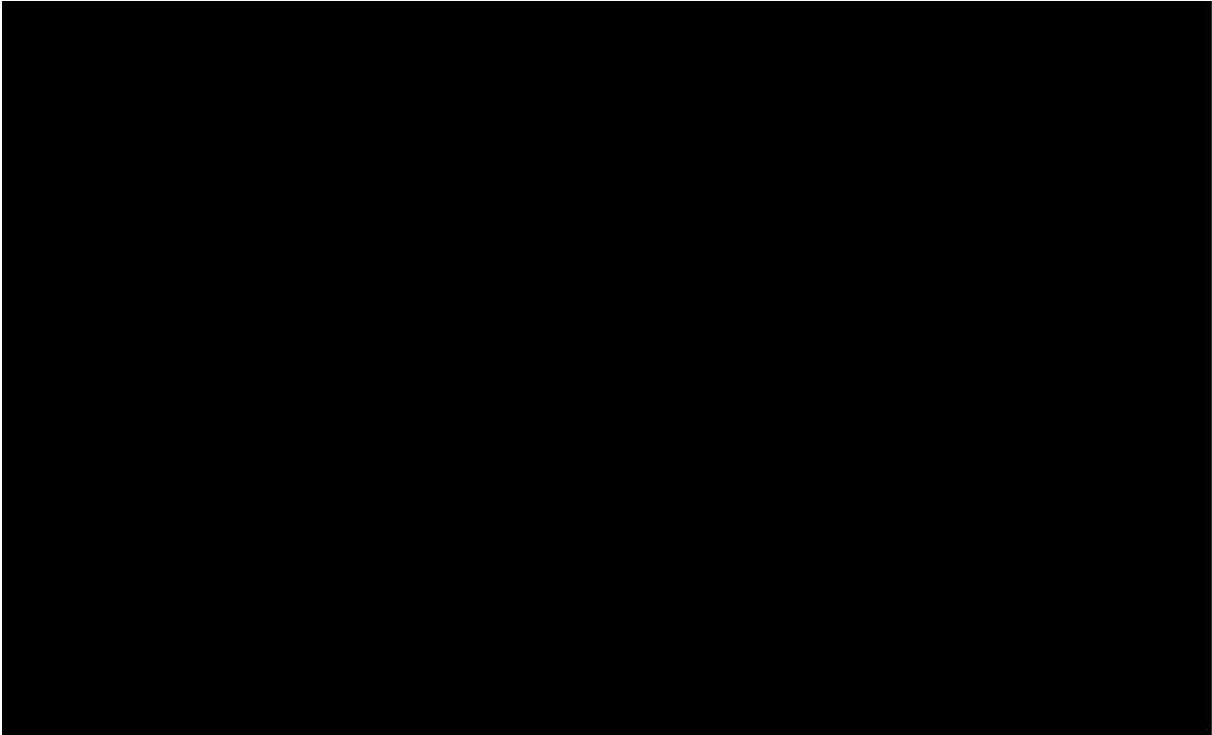
Module 2

Introducing Data Types and Operators



Table of Contents

CRITICAL SKILL 2.1: The C++ Data Types	2
--	---



Project 2-1 Talking to Mars	10
CRITICAL SKILL 2.2: Literals	12
CRITICAL SKILL 2.3: A Closer Look at Variables	15
CRITICAL SKILL 2.4: Arithmetic Operators	17
CRITICAL SKILL 2.5: Relational and Logical Operators	20
Project 2-2 Construct an XOR Logical Operation	22
CRITICAL SKILL 2.6: The Assignment Operator	25
CRITICAL SKILL 2.7: Compound Assignments	25
CRITICAL SKILL 2.8: Type Conversion in Assignments	26
CRITICAL SKILL 2.9: Type Conversion in Expressions	27
CRITICAL SKILL 2.10: Casts	27
CRITICAL SKILL 2.11: Spacing and Parentheses	28

At the core of a programming language are its data types and operators. These elements define the limits of a language and determine the kind of tasks to which it can be applied. As you might expect, C++ supports a rich assortment of both data types and operators, making it suitable for a wide range of programming. Data types and operators are a large subject. We will begin here with an examination of C++'s foundational data types and its most commonly used operators. We will also take a closer look at variables and examine the expression.

1	

Why Data Types Are Important

The data type of a variable is important because it determines the operations that are allowed and the range of values that can be stored. C++ defines several types of data, and each type has unique characteristics. Because data types differ, all variables must be declared prior to their use, and a variable declaration always includes a type specifier. The compiler requires this information in order to generate correct code. In C++ there is no concept of a “type-less” variable.

A second reason that data types are important to C++ programming is that several of the basic types are closely tied to the building blocks upon which the computer operates: bytes and words. Thus, C++ lets you operate on the same types of data as does the CPU itself. This is one of the ways that C++ enables you to write very efficient, system-level code.

CRITICAL SKILL 2.1: The C++ Data Types

C++ provides built-in data types that correspond to integers, characters, floating-point values, and Boolean values. These are the ways that data is commonly stored and manipulated by a program. As you will see later in this book, C++ allows you to construct more sophisticated types, such as classes, structures, and enumerations, but these too are ultimately composed of the built-in types.

At the core of the C++ type system are the seven basic data types shown here:

Type	Meaning
char	Character
wchar_t	Wide character
int	Integer
float	Floating point
double	Double floating point
bool	Boolean
void	Valueless

C++ allows certain of the basic types to have modifiers preceding them. A modifier alters the meaning of the base type so that it more precisely fits the needs of various situations. The data type modifiers are listed here:

signed

unsigned

long

short

The modifiers signed, unsigned, long, and short can be applied to int. The modifiers signed and unsigned can be applied to the char type. The type double can be modified by long. Table 2-1 shows all valid

2	
---	--

combinations of the basic types and the type modifiers. The table also shows the guaranteed minimum range for each type as specified by the ANSI/ISO C++ standard.

Type	Minimal Range
char	-127 to 127
unsigned char	0 to 255
signed char	-127 to 127
int	-32,767 to 32,767
unsigned int	0 to 65,535
signed int	Same as int
short int	-32,767 to 32,767
unsigned short int	0 to 65,535
signed short int	Same as short int
long int	-2,147,483,647 to 2,147,483,647
signed long int	Same as long int
unsigned long int	0 to 4,294,967,295
float	1E-37 to 1E+37, with six digits of precision
double	1E-37 to 1E+37, with ten digits of precision
long double	1E-37 to 1E+37, with ten digits of precision

Table 2-1 All Numeric Data Types Defined by C++ and Their Minimum Guaranteed Ranges as Specified by the ANSI/ISO C++ Standard

It is important to understand that minimum ranges shown in Table 2-1 are just that: minimum ranges. A C++ compiler is free to exceed one or more of these minimums, and most compilers do. Thus, the ranges of the C++ data types are implementation dependent. For example, on computers that use two's complement arithmetic (which is nearly all), an integer will have a range of at least $-32,768$ to $32,767$. In all cases, however, the range of a short int will be a subrange of an int, which will be a subrange of a long int. The same applies to float, double, and long double. In this usage, the term subrange means a range narrower than or equal to. Thus, an int and long int can have the same range, but an int cannot be larger than a long int.

Since C++ specifies only the minimum range a data type must support, you should check your compiler's documentation for the actual ranges supported. For example, Table 2-2 shows typical bit widths and ranges for the C++ data types in a 32-bit environment, such as that used by Windows XP.

Let's now take a closer look at each data type.

3	

Type	Bit Width	Typical Range
char	8	-128 to 127
unsigned char	8	0 to 255
signed char	8	-128 to 127
int	32	-2,147,483,648 to 2,147,483,647
unsigned int	32	0 to 4,294,967,295
signed int	32	-2,147,483,648 to 2,147,483,647
short int	16	-32,768 to 32,767
unsigned short int	16	0 to 65,535
signed short int	16	-32,768 to 32,767
long int	32	Same as int
signed long int	32	Same as signed int
unsigned long int	32	Same as unsigned int
float	32	1.8E-38 to 3.4E+38
double	64	2.2E-308 to 1.8E+308
long double	64	2.2E-308 to 1.8E+308
bool	N/A	True or false
wchar_t	16	0 to 65,535

Table 2-2 Typical Bit Widths and Ranges for the C++ Data Types in a 32-Bit Environment

Integers

As you learned in Module 1, variables of type `int` hold integer quantities that do not require fractional components. Variables of this type are often used for controlling loops and conditional statements, and for counting. Because they don't have fractional components, operations on `int` quantities are much faster than they are on floating-point types.

Because integers are so important to programming, C++ defines several varieties. As shown in Table 2-1, there are short, regular, and long integers. Furthermore, there are signed and unsigned versions of each. A signed integer can hold both positive and negative values. By default, integers are signed. Thus, the use of `signed` on integers is redundant (but allowed) because the default declaration assumes a signed value. An unsigned integer can hold only positive values. To create an unsigned integer, use the `unsigned` modifier.

The difference between signed and unsigned integers is in the way the high-order bit of the integer is interpreted. If a signed integer is specified, then the C++ compiler will generate code that assumes that the high-order bit of an integer is to be used as a sign flag. If the sign flag is 0, then the number is positive; if it is 1, then the number is negative. Negative numbers are almost always represented using

4	

the two's complement approach. In this method, all bits in the number (except the sign flag) are reversed, and then 1 is added to this number. Finally, the sign flag is set to 1.

Signed integers are important for a great many algorithms, but they have only half the absolute magnitude of their unsigned relatives. For example, assuming a 16-bit integer, here is 32,767:

0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

For a signed value, if the high-order bit were set to 1, the number would then be interpreted as -1 (assuming the two's complement format). However, if you declared this to be an unsigned int, then when the high-order bit was set to 1, the number would become 65,535.

To understand the difference between the way that signed and unsigned integers are interpreted by

C++, try this short program:

```
#include <iostream>
```

```
/* This program shows the difference between signed and unsigned integers. */
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    short int i; // a signed short integer  
    int j; // an unsigned short integer
```

```
    j = 60000; ←  
    i = j; ←  
    cout << i << " " << j;  
    return 0;  
}
```

60,000 is within the range of an unsigned short int, but is typically outside the range of a signed short int. Thus, it will be interpreted as a negative value when assigned to i.

The output from this program is shown here:

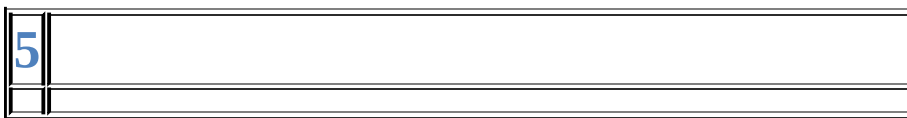
-5536 60000

These values are displayed because the bit pattern that represents 60,000 as a short unsigned integer is interpreted as -5,536 as short signed integer (assuming 16-bit short integers).

C++ allows a shorthand notation for declaring unsigned, short, or long integers. You can simply use the word unsigned, short, or long, without the int. The int is implied. For example, the following two statements both declare unsigned integer variables:

```
unsigned x;
```

```
unsigned int y;
```



Characters

Variables of type `char` hold 8-bit ASCII characters such as A, z, or G, or any other 8-bit quantity. To specify a character, you must enclose it between single quotes. Thus, this assigns X to the variable `ch`:

```
char ch;
```

```
ch = 'X';
```

You can output a `char` value using a `cout` statement. For example, this line outputs the value in `ch`:

```
cout << "This is ch: " << ch;
```

This results in the following output:

```
This is ch: X
```

The `char` type can be modified with `signed` or `unsigned`. Technically, whether `char` is signed or unsigned by default is implementation-defined. However, for most compilers `char` is signed. In these environments, the use of `signed` on `char` is also redundant. For the rest of this book, it will be assumed that `chars` are signed entities.

The type `char` can hold values other than just the ASCII character set. It can also be used as a “small” integer with the range typically from `-128` through `127` and can be substituted for an `int` when the situation does not require larger numbers. For example, the following program uses a `char` variable to control the loop that prints the alphabet on the screen:


```
// This program displays the alphabet.

#include <iostream>
using namespace std;

int main()
{
    char letter;
    for(letter = 'A'; letter <= 'Z'; letter++)
        cout << letter;

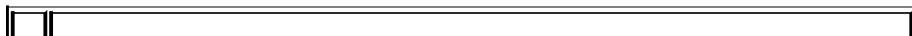
    return 0;
}
```

Use a char variable
to control a for loop.



The for loop works because the character A is represented inside the computer by the value 65, and the values for the letters A to Z are in sequential, ascending order. Thus, letter is initially set to 'A'. Each time through the loop, letter is incremented. Thus, after the first iteration, letter is equal to 'B'.

The type `wchar_t` holds characters that are part of large character sets. As you may know, many human languages, such as Chinese, define a large number of characters, more than will fit within the 8 bits provided by the `char` type. The `wchar_t` type was added to C++ to accommodate this situation. While we



6	

won't be making use of `wchar_t` in this book, it is something that you will want to look into if you are tailoring programs for the international market.



What are the seven basic types?

What is the difference between signed and unsigned integers?

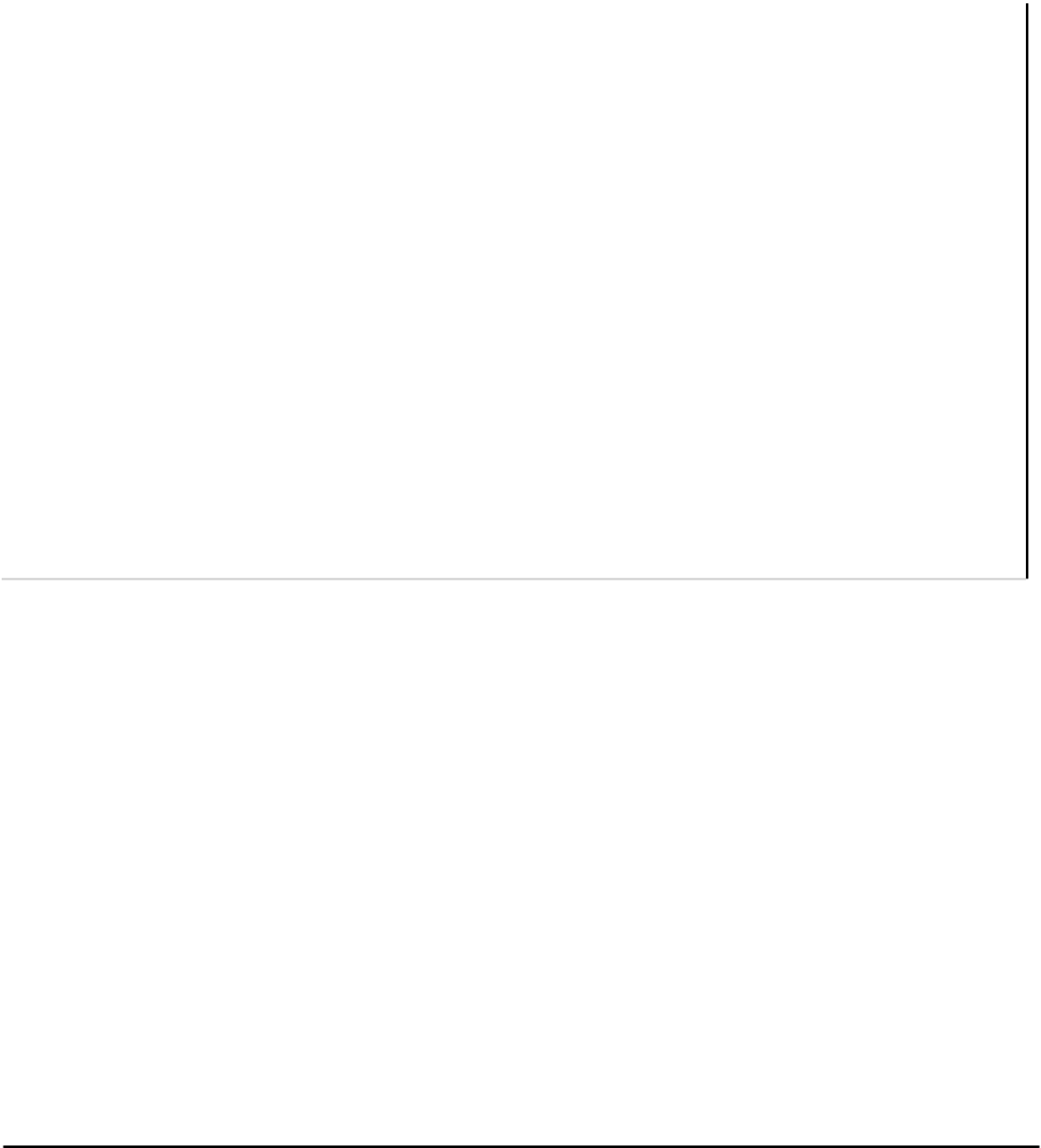
Can a char variable be used like a little integer?

Answer Key:

The seven basic types are char, wchar_t, int, float, double, bool, and void.

A signed integer can hold both positive and negative values. An unsigned integer can hold only positive values.

Yes.



Ask the Expert

Q: Why does C++ specify only minimum ranges for its built-in types rather than stating these precisely?

A: By not specifying precise ranges, C++ allows each compiler to optimize the data types for the execution environment. This is part of the reason that C++ can create high-performance software. The ANSI/ISO C++ standard simply states that the built-in types must meet certain requirements. For example, it states that an int will “have the natural size suggested by the architecture of the execution environment.” Thus, in a 32-bit environment, an int will be 32 bits long. In a 16-bit environment, an int will be 16 bits long. It would be an inefficient and unnecessary burden to force a 16-bit compiler to implement int with a 32-bit range, for example. C++’s approach avoids this. Of course, the C++ standard does specify a minimum range for the built-in types that will be available in all environments. Thus, if you write your programs in such a way that these minimal ranges are not exceeded, then your program will be portable to other environments. One last point: Each C++ compiler specifies the range

of the basic types in the header `<climits>`.

7	

Floating-Point Types

Variables of the types `float` and `double` are employed either when a fractional component is required or when your application requires very large or small numbers. The difference between a `float` and a `double` variable is the magnitude of the largest (and smallest) number that each one can hold. Typically, a `double` can store a number approximately ten times larger than a `float`. Of the two, `double` is the most commonly used. One reason for this is that many of the math functions in the C++ function library use `double` values. For example, the `sqrt()` function returns a `double` value that is the square root of its `double` argument. Here, `sqrt()` is used to compute the length of the hypotenuse given the lengths of the two opposing sides.

```
/*
    Use the Pythagorean theorem to find
    the length of the hypotenuse given
    the lengths of the two opposing sides.
*/

#include <iostream>
#include <cmath>  ← The <cmath> header is needed for the sqrt( ) function.
using namespace std;

int main() {
    double x, y, z;

    x = 5.0;
    y = 4.0;

    z = sqrt(x*x + y*y); ← The sqrt( ) function is part of C++'s math library.

    cout << "Hypotenuse is " << z;

    return 0;
}
```

The output from the program is shown here:

Hypotenuse is 6.40312

One other point about the preceding example: Because `sqrt()` is part of the C++ standard function library, it requires the standard header `<cmath>`, which is included in the program.

The long double type lets you work with very large or small numbers. It is most useful in scientific programs. For example, the long double type might be useful when analyzing astronomical data.

The bool Type

The `bool` type is a relatively recent addition to C++. It stores Boolean (that is, true/false) values. C++ defines two Boolean constants, `true` and `false`, which are the only two values that a `bool` value can have. Before continuing, it is important to understand how `true` and `false` are defined by C++. One of the fundamental concepts in C++ is that any nonzero value is interpreted as `true` and zero is `false`. This

8	

concept is fully compatible with the bool data type because when used in a Boolean expression, C++ automatically converts any nonzero value into true. It automatically converts zero into false. The reverse is also true; when used in a non-Boolean expression, true is converted into 1, and false is converted into zero. The convertibility of zero and nonzero values into their Boolean equivalents is especially important when using control statements, as you will see in Module 3. Here is a program that demonstrates the bool type:

```
Demonstrate bool values. #include <iostream>
using namespace std;

int main() {
    bool b;

    b = false;
    cout << "b is " << b << "\n";

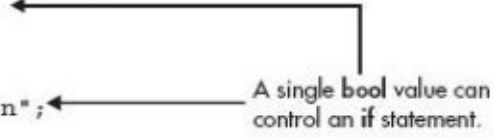
    b = true;
    cout << "b is " << b << "\n";

    // a bool value can control the if statement
    if(b) cout << "This is executed.\n";

    b = false;
    if(b) cout << "This is not executed.\n";

    // outcome of a relational operator is a true/false value
    cout << "10 > 9 is " << (10 > 9) << "\n";

    return 0;
}
```



A single bool value can control an if statement.

The diagram consists of two horizontal arrows pointing left. The top arrow originates from the text 'A single bool value can control an if statement.' and points to the 'if(b)' condition in the first if statement. The bottom arrow originates from the same text and points to the 'if(b)' condition in the second if statement. A vertical line connects the two arrows, indicating that the same concept applies to both.

The output generated by this program is shown here:

b is 0

b is 1

This is executed.

10 > 9 is 1

There are three interesting things to notice about this program. First, as you can see, when a bool value is output using cout, 0 or 1 is displayed. As you will see later in this book, there is an output option that causes the words “false” and “true” to be displayed.

Second, the value of a bool variable is sufficient, by itself, to control the if statement. There is no need to write an if statement like this:

if(b == true) ...

9	

Third, the outcome of a relational operator, such as $<$, is a Boolean value. This is why the expression $10 > 9$ displays the value 1. Further, the extra set of parentheses around $10 > 9$ is necessary because the $<<$ operator has a higher precedence than the $>$.

void

The void type specifies a valueless expression. This probably seems strange now, but you will see how void is used later in this book.



What is the primary difference between float and double?

What values can a bool variable have? To what Boolean value does zero convert?

What is void?

Answer Key:

The primary difference between float and double is in the magnitude of the values they can hold.

Variables of type bool can be either true or false. Zero converts to false.

void is a type that stands for valueless.

Project 2-1 Talking to Mars

At its closest point to Earth, Mars is approximately 34,000,000 miles away. Assuming there is someone on Mars that you want to talk with, what is the delay between the time a radio signal leaves Earth and the time it arrives on Mars? This project creates a program that answers this question. Recall that radio signals travel at the speed of light, approximately 186,000 miles per second. Thus, to compute the delay, you will need to divide the distance by the speed of light. Display the delay in terms of seconds and also in

minutes.

Step by Step

Create a new file called Mars.cpp.

To compute the delay, you will need to use floating-point values. Why? Because the time interval will have a fractional component. Here are the variables used by the program:

```
double distance;
```

10	

```
double lightspeed;  
  
double delay;  
  
double delay_in_min;
```

Give distance and lightspeed initial values, as shown here:

```
distance = 34000000.0; // 34,000,000 miles  
lightspeed = 186000.0; // 186,000 per second
```

To compute the delay, divide distance by lightspeed. This yields the delay in seconds. Assign this value to delay and display the results. These steps are shown here:

```
delay = distance / lightspeed;  
  
cout << "Time delay when talking to Mars: " << delay  
<< " seconds.\n";
```

Divide the number of seconds in delay by 60 to obtain the delay in minutes; display that result using these lines of code:

```
delay_in_min = delay / 60.0;
```

Here is the entire Mars.cpp program listing:

```
/*  
  
Project 2-1 Talking to Mars */  
  
#include <iostream> using namespace std;  
  
int main()  
{  
  
    double distance;  
  
    double lightspeed;
```

```
double delay;

double delay_in_min;

distance = 34000000.0; // 34,000,000
miles lightspeed = 186000.0; // 186,000
per second

delay = distance / lightspeed;

cout << "Time delay when talking to Mars: " << delay
<< " seconds.\n";

delay_in_min = delay / 60.0;

cout << "This is " << delay_in_min << "
minutes.";
```

```
        return 0;  
    }
```

Compile and run the program. The following result is displayed:

```
Time delay when talking to Mars: 182.796 seconds.  
This is 3.04659 minutes.
```

On your own, display the time delay that would occur in a bidirectional conversation with Mars.

CRITICAL SKILL 2.2: Literals

Literals refer to fixed, human-readable values that cannot be altered by the program. For example, the value 101 is an integer literal. Literals are also commonly referred to as constants. For the most part, literals and their usage are so intuitive that they have been used in one form or another by all the preceding sample programs. Now the time has come to explain them formally.

C++ literals can be of any of the basic data types. The way each literal is represented depends upon its type. As explained earlier, character literals are enclosed between single quotes. For example, 'a' and '%' are both character literals.

Integer literals are specified as numbers without fractional components. For example, 10 and -100 are integer constants. Floating-point literals require the use of the decimal point followed by the number's fractional component. For example, 11.123 is a floating-point constant. C++ also allows you to use scientific notation for floating-point numbers.

All literal values have a data type, but this fact raises a question. As you know, there are several different types of integers, such as int, short int, and unsigned long int. There are also three different floating-point types: float, double, and long double. The question is: How does the compiler determine the type of a literal? For example, is 123.23 a float or a double? The answer to this question has two parts. First, the C++ compiler automatically makes

certain assumptions about the type of a literal and, second, you can explicitly specify the type of a literal, if you like.

By default, the C++ compiler fits an integer literal into the smallest compatible data type that will hold it, beginning with int. Therefore, assuming 16-bit integers, 10 is int by default, but 103,000 is long. Even though the value 10 could be fit into a char, the compiler will not do this because it means crossing type boundaries.

By default, floating-point literals are assumed to be double. Thus, the value 123.23 is of type double.

For virtually all programs you will write as a beginner, the compiler defaults are perfectly adequate. In cases where the default assumption that C++ makes about a numeric literal is not what you want, C++ allows you to specify the exact type of numeric literal by using a suffix. For floating-point types, if you follow the number with an F, the number is treated as a float. If you follow it with an L, the number becomes a long double. For integer types, the U suffix stands for unsigned and the L for long. (Both the U and the L must be used to specify an unsigned long.) Some examples are shown here:

12	

Data Type	Examples of Constants
int	1 123 21000 -234
long int	35000L -34L
unsigned int	10000U 987U 40000U
unsigned long	12323UL 900000UL
float	123.23F 4.34e-3F
double	23.23 123123.33 -0.9876324
long double	1001.2L

Hexadecimal and Octal Literals

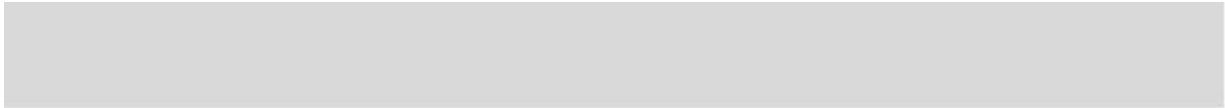
As you probably know, in programming it is sometimes easier to use a number system based on 8 or 16 instead of 10. The number system based on 8 is called octal, and it uses the digits 0 through 7. In octal, the number 10 is the same as 8 in decimal. The base-16 number system is called hexadecimal and uses the digits 0 through 9 plus the letters A through F, which stand for 10, 11, 12, 13, 14, and 15. For example, the hexadecimal number 10 is 16 in decimal. Because of the frequency with which these two number systems are used, C++ allows you to specify integer literals in hexadecimal or octal instead of decimal. A hexadecimal literal must begin with 0x (a zero followed by an x). An octal literal begins with a zero. Here are some examples:

```
hex = 0xFF; // 255 in decimal
```

```
oct = 011; // 9 in decimal
```

String Literals

C++ supports one other type of literal in addition to those of the predefined data types: the string. A string is a set of characters enclosed by double quotes. For example, “this is a test” is a string. You have seen examples of strings in some of the cout statements in the preceding sample programs. Keep in mind one important fact: although C++ allows you to define string constants, it does not have a built-in string data type. Instead, as you will see a little later in this book, strings are supported in C++ as character arrays. (C++ does, however, provide a string type in its class library.)



Ask the Expert

Q: You showed how to specify a char literal. Is a wchar_t literal specified in the same way?

A: No. A wide-character constant (that is, one that is of type wchar_t) is preceded with the character L.

For example:

```
wchar_t wc;
```

```
wc = L'A';
```

Here, `wc` is assigned the wide-character constant equivalent of `A`. You will not use wide characters often in your normal day-to-day programming, but they are something that might be of importance if you need to internationalize your program.

Character Escape Sequences

Enclosing character constants in single quotes works for most printing characters, but a few characters, such as the carriage return, pose a special problem when a text editor is used. In addition, certain other characters, such as the single and double quotes, have special meaning in C++, so you cannot use them directly. For these reasons, C++ provides the character escape sequences, sometimes referred to as backslash character constants, shown in Table 2-3, so that you can enter them into a program. As you can see, the `\n` that you have been using is one of the escape sequences.

Code	Meaning
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\"	Double quote
\'	Single quote character
\\	Backslash
\v	Vertical tab
\a	Alert
\?	?
\N	Octal constant (where <i>N</i> is an octal constant)
\xN	Hexadecimal constant (where <i>N</i> is a hexadecimal constant)

Table 2-3 The Character Escape Sequences



Ask the Expert

Q: Is a string consisting of a single character the same as a character literal? For example, is “k” the same as ‘k’?

A: No. You must not confuse strings with characters. A character literal represents a single letter of type `char`. A string containing only one letter is still a string. Although strings consist of characters, they are not the same type.

The following sample program illustrates a few of the escape sequences:

```
// Demonstrate some escape sequences.

#include <iostream>
using namespace std;

int main()
{
    cout << "one\ttwo\tthree\n";
    cout << "123\b\b45"; ← The \b\b will backspace over the 2 and 3.

    return 0;
}
```

The output is shown here:

```
one      two      three
145
```

Here, the first cout statement uses tabs to position the words “two” and “three”. The second cout statement displays the characters 123. Next, two backspace characters are output, which deletes the 2 and 3. Finally, the characters 4 and 5 are displayed.



By default, what is the type of the literal 10? What is the type of the literal 10.0?

How do you specify 100 as a long int?
How do you specify 100 as an unsigned int?

What is `\b`?

100 as a long int is 100L. 100 as an unsigned int is 100U.

\b is the escape sequence that causes a backspace.

Variables were introduced in Module 1. Here we will take a closer look at them. As you learned, variables are declared using this form of statement:

15	

type var-name;

where type is the data type of the variable and var-name is its name. You can declare a variable of any valid type. When you create a variable, you are creating an instance of its type. Thus, the capabilities of a variable are determined by its type. For example, a variable of type bool stores Boolean values. It cannot be used to store floating-point values. Furthermore, the type of a variable cannot change during its lifetime. An int variable cannot turn into a double variable, for example.

Initializing a Variable

You can assign a value to a variable at the same time that it is declared. To do this, follow the variable's name with an equal sign and the value being assigned. This is called a variable initialization. Its general form is shown here:

type var = value;

Here, value is the value that is given to var when var is created.

Here are some examples:

```
int count = 10; // give count an initial value of 10
char ch = 'X'; // initialize ch with the letter X
float f = 1.2F; // f is initialized with 1.2
```

When declaring two or more variables of the same type using a comma separated list, you can give one or more of those variables an initial value. For example,

```
int a, b = 8, c = 19, d; // b and c have initializations
// In this case, only b and c are initialized.
```

Dynamic Initialization

Although the preceding examples have used only constants as initializers, C++ allows variables to be initialized dynamically, using any expression valid at the time the variable is declared. For example, here is a short program that computes the volume of a cylinder given the radius of its base and its height:

16	

```

// Demonstrate dynamic initialization.

#include <iostream>
using namespace std;

int main() {
    double radius = 4.0, height = 5.0;

    // dynamically initialize volume
    double volume = 3.1416 * radius * radius * height;
    cout << "Volume is " << volume;
    return 0;
}

```

← volume is dynamically initialized at runtime.

Here, three local variables—radius, height, and volume—are declared. The first two, radius and height, are initialized by constants. However, volume is initialized dynamically to the volume of the cylinder. The key point here is that the initialization expression can use any element valid at the time of the initialization, including calls to functions, other variables, or literals.

Operators

C++ provides a rich operator environment. An operator is a symbol that tells the compiler to perform a specific mathematical or logical manipulation. C++ has four general classes of operators: arithmetic,

bitwise, relational, and logical. C++ also has several additional operators that handle certain special situations. This chapter will examine the arithmetic, relational, and logical operators. We will also examine the assignment operator. The bitwise and other special operators are examined later.

CRITICAL SKILL 2.4: Arithmetic Operators

C++ defines the following arithmetic operators:

Operator	Meaning
+	Addition
-	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
--	Decrement

The operators +, -, *, and / all work the same way in C++ as they do in algebra. These can be applied to any built-in numeric data type. They can also be applied to values of type char.



The % (modulus) operator yields the remainder of an integer division. Recall that when / is applied to an integer, any remainder will be truncated; for example, 10/3 will equal 3 in integer division. You can obtain the remainder of this division by using the % operator. For example, 10 % 3 is 1. In C++, the % can be applied only to integer operands; it cannot be applied to floating-point types.

The following program demonstrates the modulus operator:

```
// Demonstrate the modulus operator.
```

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    int x, y;
```

```
    x = 10;
```

```
    y = 3;
```

```
    cout << x << " / " << y << " is " << x / y <<
```

```
        with a remainder of "  
    << x % y << "\n";
```

```
    = 1;
```

```
    = 2;
```

```
    cout << x << " / " << y << " is " << x / y << "\n"  
    << x << " % " << y << " is " << x % y;
```

```
    return 0;
```

```
}
```

The output is shown here:

$10 / 3$ is 3 with a remainder of 1

$1 / 2$ is 0

$1 \% 2$ is 1

Increment and Decrement

Introduced in Module 1, the `++` and the `--` are the increment and decrement operators. They have some special properties that make them quite interesting. Let's begin by reviewing precisely what the increment and decrement operators do.

The increment operator adds 1 to its operand, and the decrement operator subtracts 1. Therefore,

`x = x + 1;`

is the same as

`x++;`

and

```
x = x - 1;
```

is the same as

```
--x;
```

Both the increment and decrement operators can either precede (prefix) or follow (postfix) the operand. For example,

```
x = x + 1;
```

can be written as

```
++x; // prefix form
```

or as

```
x++; // postfix form
```

In this example, there is no difference whether the increment is applied as a prefix or a postfix. However, when an increment or decrement is used as part of a larger expression, there is an important difference. When an increment or decrement operator precedes its operand, C++ will perform the operation prior to obtaining the operand's value for use by the rest of the expression. If the operator follows its operand, then C++ will obtain the operand's value before incrementing or decrementing it. Consider the following:

```
x = 10; y = ++x;
```

In this case, y will be set to 11. However, if the code is written as

```
x = 10; y = x++;
```

then y will be set to 10. In both cases, x is still set to 11; the difference is when it happens. There are significant advantages in being able to control when the increment or decrement operation takes place.

The precedence of the arithmetic operators is shown here:

Highest	++ --
	- (unary minus)
	* / %
Lowest	+ -

Operators on the same precedence level are evaluated by the compiler from left to right. Of course, parentheses may be used to alter the order of evaluation. Parentheses are treated by C++ in the same way that they are by virtually all other computer languages: they force an operation, or a set of operations, to have a higher precedence level.

19	



Ask the Expert

Q:

A:

Does the increment operator ++ have anything to do with the name C++?

Yes! As you know, C++ is built upon the C language. C++ adds to C several enhancements, most of

which support object-oriented programming. Thus, C++ represents an incremental improvement to C, and the addition of the ++ (which is, of course, the increment operator) to the name C is a fitting way to describe C++.

Stroustrup initially named C++ “C with Classes,” but at the suggestion of Rick Mascitti, he later changed the name to C++. While the new language was already destined for success, the adoption of the name C++ virtually guaranteed its place in history because it was a name that every C programmer would instantly recognize!

CRITICAL SKILL 2.5: Relational and Logical Operators

In the terms relational operator and logical operator, relational refers to the relationships that values can have with one another, and logical refers to the ways in which true and false values can be connected together. Since the relational operators produce true or false results, they often work with the logical operators. For this reason, they will be discussed together here.

The relational and logical operators are shown in Table 2-4. Notice that in C++, not equal to is represented by `!=` and equal to is represented by the double equal sign, `==`. In C++, the outcome of a relational or logical expression produces a bool result. That is, the outcome of a relational or logical expression is either true or false.

NOTE: For older compilers, the outcome of a relational or logical expression will be an integer value of either 0 or 1. This difference is mostly academic, though, because C++ automatically converts **true** into 1 and **false** into 0, and vice versa as explained earlier.

The operands for a relational operator can be of nearly any type as long as they can be meaningfully compared. The operands to the logical operators must produce a true or false result. Since any nonzero value is true and zero is false, this means that the logical operators can be used with any expression that evaluates to a zero or nonzero result. Thus, any expression other than one that has a void result can be used.

Relational Operators	
Operator	Meaning
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
==	Equal to
!=	Not equal to
Logical Operators	
Operator	Meaning
&&	AND
	OR
!	NOT

Table 2-4 The Relational and Logical Operators in C++

The logical operators are used to support the basic logical operations AND, OR, and NOT, according to the following truth table:

p	q	p AND q	p OR q	NOT p
False	False	False	False	True
False	True	False	True	True
True	True	True	True	False
True	False	False	True	False

Here is a program that demonstrates several of the relational and logical operators:

```
// Demonstrate the relational and logical operators.
```

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    int i, j;
```

```
    bool b1, b2;
```

```
    i = 10;
```

```
    j = 11;
```

```
    if(i < j) cout << "i < j\n";
```

```
    if(i <= j) cout << "i <= j\n";
```

```

        if(i != j) cout << "i != j\n";

        if(i == j) cout << "this won't execute\n"; if(i >= j)
        cout << "this won't execute\n"; if(i > j) cout <<
        "this won't execute\n";

        b1 = true;

        b2 = false;

        if(b1 && b2) cout << "this won't execute\n"; if(!
        (b1 && b2)) cout << "!(b1 && b2) is true\n"; if(b1
        || b2) cout << "b1 || b2 is true\n";

        return 0;

    }

```

The output from the program is shown here:

```

i < j
i <= j
i != j
!(b1 && b2) is true
b1 || b2 is true

```

Both the relational and logical operators are lower in precedence than the arithmetic operators. This means that an expression like $10 > 1+12$ is evaluated as if it were written $10 > (1+12)$. The result is, of course, false.

You can link any number of relational operations together using logical operators. For example, this expression joins three relational operations:

```
var > 15 || !(10 < count) && 3 <= item
```

The following table shows the relative precedence of the relational and logical operators:

Highest	!
	> >= < <=
	== !=
	&&
Lowest	

Project 2-2 Construct an XOR Logical Operation

C++ does not define a logical operator that performs an exclusive-OR operation, usually referred to as XOR. The XOR is a binary operation that yields true when one and only one operand is true. It has this truth table:

22	

p	q	p XOR q
False	False	False
False	True	True
True	False	True
True	True	False

Some programmers have called the omission of the XOR a flaw. Others argue that the absence of the XOR logical operator is simply part of C++'s streamlined design, which avoids redundant features. They point out that it is easy to create an XOR logical operation using the three logical operators that C++ does provide.

In this project, you will construct an XOR operation using the `&&`, `||`, and `!` operators. You can decide for yourself if the omission of an XOR logical operator is a design flaw or an elegant feature!

Step by Step

Create a new file called XOR.cpp.

Assuming two Boolean values, p and q, a logical XOR is constructed like this:

```
(p || q) && !(p && q)
```

Let's go through this carefully. First, p is ORed with q. If this result is true, then at least one of the operands is true. Next, p is ANDed with q. This result is true if both operands are true. This result is then inverted using the NOT operator. Thus, the outcome of `!(p && q)` will be true when either p, q, or both are false. Finally, this result is ANDed with the result of `(p || q)`. Thus, the

entire expression will be true when one but not both operands is true.

Here is the entire XOR.cpp program listing. It demonstrates the XOR operation for all four possible combinations of true/false values.

```
/*
```

Project 2-2

Create an XOR using the C++ logical operators.

```
*/
```

```
#include <iostream> #include <cmath> using  
namespace std;
```

```
int main()
```

```
{
```

```
    bool p, q;
```

```
    p = true;
```

```

    q = true;

    cout << p << " XOR " << q << " is " <<
    ( (p || q) && !(p && q) ) << "\n";

    p = false;

    q = true;

    cout << p << " XOR " << q << " is " <<
    ( (p || q) && !(p && q) ) << "\n";

    p = true;

    q = false;

    cout << p << " XOR " << q << " is " <<
    ( (p || q) && !(p && q) ) << "\n";

    p = false;

    q = false;

    cout << p << " XOR " << q << " is " <<
    ( (p || q) && !(p && q) ) << "\n";

    return 0;

}

```

Compile and run the program. The following output is produced:

1 XOR 1 is 0

0 XOR 1 is 1

1 XOR 0 is 1

0 XOR 0 is 0

Notice the outer parentheses surrounding the XOR operation inside the cout statements. They are necessary because of the precedence of C++'s operators. The << operator is higher in precedence than the logical operators. To prove this, try removing the outer parentheses, and then attempt to compile the program. As you will see, an error will be reported.



Progress Check

1. What does the % operator do? To what types can it be applied?

24



--

```
int index = 10;
```

The result of a relational or logical expression is of type `bool`.

CRITICAL SKILL 2.6: The Assignment Operator

You have been using the assignment operator since Module 1. Now it is time to take a formal look at it. The assignment operator is the single equal sign, =. The assignment operator works in C++ much as it does in any other computer language. It has this general form:

```
var = expression;
```

Here, the value of the expression is given to var. The assignment operator does have one interesting attribute: it allows you to create a chain of assignments. For example, consider this fragment:

```
int x, y, z;  
  
x = y = z = 100; // set x, y, and z to 100
```

This fragment sets the variables x, y, and z to 100 using a single statement. This works because the = is an operator that yields the value of the right-hand expression. Thus, the value of z = 100 is 100, which is then assigned to y, which in turn is assigned to x. Using a “chain of assignment” is an easy way to set a group of variables to a common value.

CRITICAL SKILL 2.7: Compound Assignments

C++ provides special compound assignment operators that simplify the coding of certain assignment statements. Let’s begin with an example. The assignment statement shown here:

```
x = x + 10;
```

can be written using a compound assignment as

`x += 10;`

The operator pair `+=` tells the compiler to assign to `x` the value of `x` plus 10. Here is another example. The statement

`x = x - 100;`

25	

is the same as

```
x -= 100;
```

Both statements assign to *x* the value of *x* minus 100. There are compound assignment operators for most of the binary operators (that is, those that require two operands). Thus, statements of the form

```
var = var op expression;
```

can be converted into this compound form:

```
var op = expression;
```

Because the compound assignment statements are shorter than their noncompound equivalents, the compound assignment operators are also sometimes called the shorthand assignment operators.

The compound assignment operators provide two benefits. First, they are more compact than their “longhand” equivalents. Second, they can result in more efficient executable code (because the operand is evaluated only once). For these reasons, you will often see the compound assignment operators used in professionally written C++ programs.

CRITICAL SKILL 2.8: Type Conversion in Assignments

When variables of one type are mixed with variables of another type, a type conversion will occur. In an assignment statement, the type conversion rule is easy: The value of the right side (expression side) of the assignment is converted to the type of the left side (target variable), as illustrated here:

```
int x;
```

```
char ch;
```

```
float f;
```

```
ch = x; /* line 1 */
```

```
x = f; /* line 2 */
```

```
f = ch; /* line 3 */
```

```
f = x; /* line 4 */
```

In line 1, the high-order bits of the integer variable *x* are lopped off, leaving *ch* with the lower 8 bits. If *x* were between -128 and 127 , *ch* and *x* would have identical values. Otherwise, the value of *ch* would reflect only the lower-order bits of *x*. In line 2, *x* will receive the nonfractional part of *f*. In line 3, *f* will convert the 8-bit integer value stored in *ch* to the same value in the floating-point format. This also happens in line 4, except that *f* will convert an integer value into floating-point format.

When converting from integers to characters and long integers to integers, the appropriate number of high-order bits will be removed. In many 32-bit environments, this means that 24 bits will be lost when going from an integer to a character, and 16 bits will be lost when going from an integer to a short integer. When converting from a floating-point type to an integer, the fractional part will be lost. If the target type is not large enough to store the result, then a garbage value will result.

26	

A word of caution: Although C++ automatically converts any built-in type into another, the results won't always be what you want. Be careful when mixing types in an expression.

Expressions

Operators, variables, and literals are constituents of expressions. You might already know the general form of an expression from other programming experience or from algebra. However, a few aspects of expressions will be discussed now.

CRITICAL SKILL 2.9: Type Conversion in Expressions

When constants and variables of different types are mixed in an expression, they are converted to the same type. First, all char and short int values are automatically elevated to int. This process is called integral promotion. Next, all operands are converted “up” to the type of the largest operand, which is called type promotion. The promotion is done on an operation-by-operation basis. For example, if one operand is an int and the other a long int, then the int is promoted to long int. Or, if either operand is a double, the other operand is promoted to double. This means that conversions such as that from a char to a double are perfectly valid. Once a conversion has been applied, each pair of operands will be of the same type, and the result of each operation will be the same as the type of both operands.

Converting to and from bool

As mentioned earlier, values of type bool are automatically converted into the integers 0 or 1 when used in an integer expression. When an integer result is converted to type bool, 0 becomes false and nonzero becomes true. Although bool is a fairly recent addition to C++, the automatic conversions to and from integers mean that it has virtually no impact on older code. Furthermore, the automatic conversions allow C++ to maintain its original definition of true and false as zero and nonzero.

CRITICAL SKILL 2.10: Casts

It is possible to force an expression to be of a specific type by using a construct called a cast. A cast is an explicit type conversion. C++ defines five types of casts. Four allow detailed and sophisticated control over casting and are described later in this book after objects have been explained. However, there is one type of cast that you can use now. It is C++'s most general cast because it can transform any type into any other type. It was also the only type of cast that early versions of C++ supported. The general form of this cast is

(type) expression

where type is the target type into which you want to convert the expression. For example, if you wish to make sure the expression $x/2$ is evaluated to type float, you can write

`(float) x / 2`

Casts are considered operators. As an operator, a cast is unary and has the same precedence as any other unary operator.

27	

There are times when a cast can be very useful. For example, you may wish to use an integer for loop control, but also perform computation on it that requires a fractional part, as in the program shown here:

```
// Demonstrate a cast.

#include <iostream>
using namespace std;

int main(){
    int i;

    for(i=1; i <= 10; ++i )
        cout << i << "/ 2 is: " << (float) i / 2 << '\n';

    return 0;
}
```

The cast to `float` causes a fractional component to be displayed.

Here is the output from this program:

1/ 2 is: 0.5

2/ 2 is: 1

3/ 2 is: 1.5

4/ 2 is: 2

5/ 2 is: 2.5

6/ 2 is: 3

7/ 2 is: 3.5

8/ 2 is: 4

9/ 2 is: 4.5

10/ 2 is: 5

Without the cast (float) in this example, only an integer division would be performed. The cast ensures that the fractional part of the answer will be displayed.

CRITICAL SKILL 2.11: Spacing and Parentheses

An expression in C++ can have tabs and spaces in it to make it more readable. For example, the following two expressions are the same, but the second is easier to read:

```
x=10/y*(127/x);
```

```
x = 10 / y * (127/x);
```

Parentheses increase the precedence of the operations contained within them, just like in algebra. Use of redundant or additional parentheses will not cause errors or slow down the execution of the expression. You are encouraged to use parentheses to make clear the exact order of evaluation, both for yourself and for others who may have to figure out your program later. For example, which of the following two expressions is easier to read?

28	

```
x = y/3-34*temp+127;
```

```
x = (y/3) - (34*temp) + 127;
```

Project 2-3 Compute the Regular Payments on a Loan

In this project, you will create a program that computes the regular payments on a loan, such as a car loan. Given the principal, the length of time, number of payments per year, and the interest rate, the program will compute the payment. Since this is a financial calculation, you will need to use floating-point data types for the computations. Since `double` is the most commonly used floating-point type, we will use it in this project. This project also demonstrates another C++ library function: `pow()`.

To compute the payments, you will use the following formula:

$$\text{Payment} = \frac{\text{IntRate} * (\text{Principal} / \text{PayPerYear})}{1 - ((\text{IntRate} / \text{PayPerYear}) + 1)^{-\text{PayPerYear} * \text{NumYears}}}$$

where `IntRate` specifies the interest rate, `Principal` contains the starting balance, `PayPerYear` specifies the number of payments per year, and `NumYears` specifies the length of the loan in years.

Notice that in the denominator of the formula, you must raise one value to the power of another. To do this, you will use `pow()`. Here is how you will call it:

```
result = pow(base, exp);
```

`pow()` returns the value of `base` raised to the `exp` power. The arguments to

`pow()` are double values, and `pow()` returns a value of type double.

Step by Step

Create a new file called `RegPay.cpp`.

Here are the variables that will be used by the program:

```
double Principal; // original principal
```

```
double IntRate; // interest rate, such as 0.075 double  
PayPerYear; // number of payments per year double  
NumYears; // number of years
```

```
double Payment; // the regular payment
```

```
double numer, denom; // temporary work variables  
double b, e; // base and exponent for call to pow()
```

Notice how each variable declaration is followed by a comment that describes its use. This helps anyone reading your program understand the purpose of each variable. Although we won't include

29	

such detailed comments for most of the short programs in this book, it is a good practice to follow as your programs become longer and more complicated.

Add the following lines of code, which input the loan information:

```
cout << "Enter principal: "; cin >> Principal;
```

```
cout << "Enter interest rate (i.e., 0.075): "; cin >> IntRate;
```

```
cout << "Enter number of payments per year: "; cin >> PayPerYear;
```

```
cout << "Enter number of years: "; cin >> NumYears;
```

Add the lines that perform the financial calculation:

```
numer = IntRate * Principal / PayPerYear;
```

```
e = -(PayPerYear * NumYears); b = (IntRate / PayPerYear) + 1; denom = 1 - pow(b, e); Payment = numer / denom;
```

Finish the program by outputting the regular payment, as shown here: `cout << "Payment is " << Payment;`

Here is the entire RegPay.cpp program listing:

```
/*
```

Project 2-3

Compute the regular payments for a loan. Call this file RegPay.cpp

```
*/
```

```
#include <iostream> #include <cmath> using
```

```
namespace std;

int main() {

    double Principal; // original principal

    double IntRate; // interest rate, such as
    0.075 double PayPerYear; // number of
    payments per year double NumYears; //
    number of years
```

30	

```

double Payment; // the regular payment

double numer, denom; // temporary work
variables double b, e; // base and
exponent for call to pow()

cout << "Enter principal: ";

cin >> Principal;

cout << "Enter interest rate (i.e., 0.075):
"; cin >> IntRate;

cout << "Enter number of payments per
year: "; cin >> PayPerYear;

cout << "Enter number of years: ";

cin >> NumYears;

numer = IntRate * Principal /
PayPerYear;

e = -(PayPerYear * NumYears);

b = (IntRate / PayPerYear) + 1;

denom = 1 - pow(b, e);

Payment = numer / denom;

cout << "Payment is " << Payment;

return 0;

}

```

Here is a sample run:

Enter principal: 10000

Enter interest rate (i.e., 0.075): 0.075

Enter number of payments per year: 12

Enter number of years: 5

Payment is 200.379

On your own, have the program display the total amount of interest paid over the life of the loan.

What type of integers are supported by C++?

By default, what type is 12.2?



Module 2 Mastery Check

31	

What values can a bool variable have?

What is the long integer data type?

What escape sequence produces a tab? What escape sequence rings the bell?

A string is surrounded by double quotes. True or false?

What are the hexadecimal digits?

Show the general form for initializing a variable when it is declared.

What does the % do? Can it be used on floating-point values?

Explain the difference between the prefix and postfix forms of the increment operator.

Which of the following are logical operators in C++?

&&

##

||

\$\$

!

How can

`x = x + 12;`

be rewritten?

What is a cast?

Write a program that finds all of the prime numbers between 1 and 100.

32	

CRITICAL SKILL 6.5: Returning References

A function can return a reference. In C++ programming, there are several uses for reference return values. Some of these uses must wait until later in this book. However, there are some that you can use now.

When a function returns a reference, it returns an implicit pointer to its return value. This gives rise to a rather startling possibility: the function can be used on the left side of an assignment statement! For example, consider this simple program:

[illegible]

9	

The output of this program is shown here:

```
100
100
99.1
```

Let's examine this program closely. At the beginning, `f()` is declared as returning a reference to a double, and the global variable `val` is initialized to 100. In `main()`, the following statement displays the original value of `val`:

```
cout << f() << '\n'; // display val's value
```

When `f()` is called, it returns a reference to `val` using this return statement:

```
return val; // return reference to val
```

This statement automatically returns a reference to `val` rather than `val`'s value. This reference is then used by the `cout` statement to display `val`'s value.

In the line

```
x = f(); // assign value of val to x
```

the reference to `val` returned by `f()` assigns the value of `val` to `x`. The most interesting line in the program is shown here:

```
f() = 99.1; // change val's value
```

This statement causes the value of `val` to be changed to 99.1. Here is why: since `f()` returns a reference to `val`, this reference becomes the target of the assignment statement. Thus, the value of 99.1 is assigned to `val` indirectly, through the reference to it returned by `f()`.

Here is another sample program that uses a reference return type:

10	

```

// Return a reference to an array element.

#include <iostream>
using namespace std;

double &change_it(int i); // return a reference

double vals[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };

int main()
{
    int i;

    cout << "Here are the original values: ";
    for(i=0; i < 5; i++)
        cout << vals[i] << " ";
    cout << "\n";

    change_it(1) = 5298.23; // change 2nd element
    change_it(3) = -98.8; // change 4th element

    cout << "Here are the changed values: ";
    for(i=0; i < 5; i++)
        cout << vals[i] << " ";
    cout << "\n";

    return 0;
}

double &change_it(int i)
{
    return vals[i]; // return a reference to the ith element
}

```

This program changes the values of the second and fourth elements in the vals array. The program displays the following output:

Here are the original values: 1.1 2.2 3.3 4.4 5.5

Here are the changed values: 1.1 5298.23 3.3 -98.8 5.5

Let's see how this is accomplished.

The `change_it()` function is declared as returning a reference to a double. Specifically, it returns a reference to the element of `vals` that is specified by its parameter `i`. The reference returned by `change_it()` is then used in `main()` to assign a value to that element.

When returning a reference, be careful that the object being referred to does not go out of scope. For example, consider this function:

11	

```
// Error, cannot return reference to local var.  
int &f()  
{  
    int i = 10;  
  
    return i; ← Error! i will go out of scope  
              when f() returns.  
}
```

In `f()`, the local variable `i` will go out of scope when the function returns. Therefore, the reference to `i` returned by `f()` will be undefined. Actually, some compilers will not compile `f()` as written for precisely this reason. However, this type of problem can be created indirectly, so be careful which object you return a reference to.

Module

Inheritance, Virtual
Functions,
and
Polymorphism

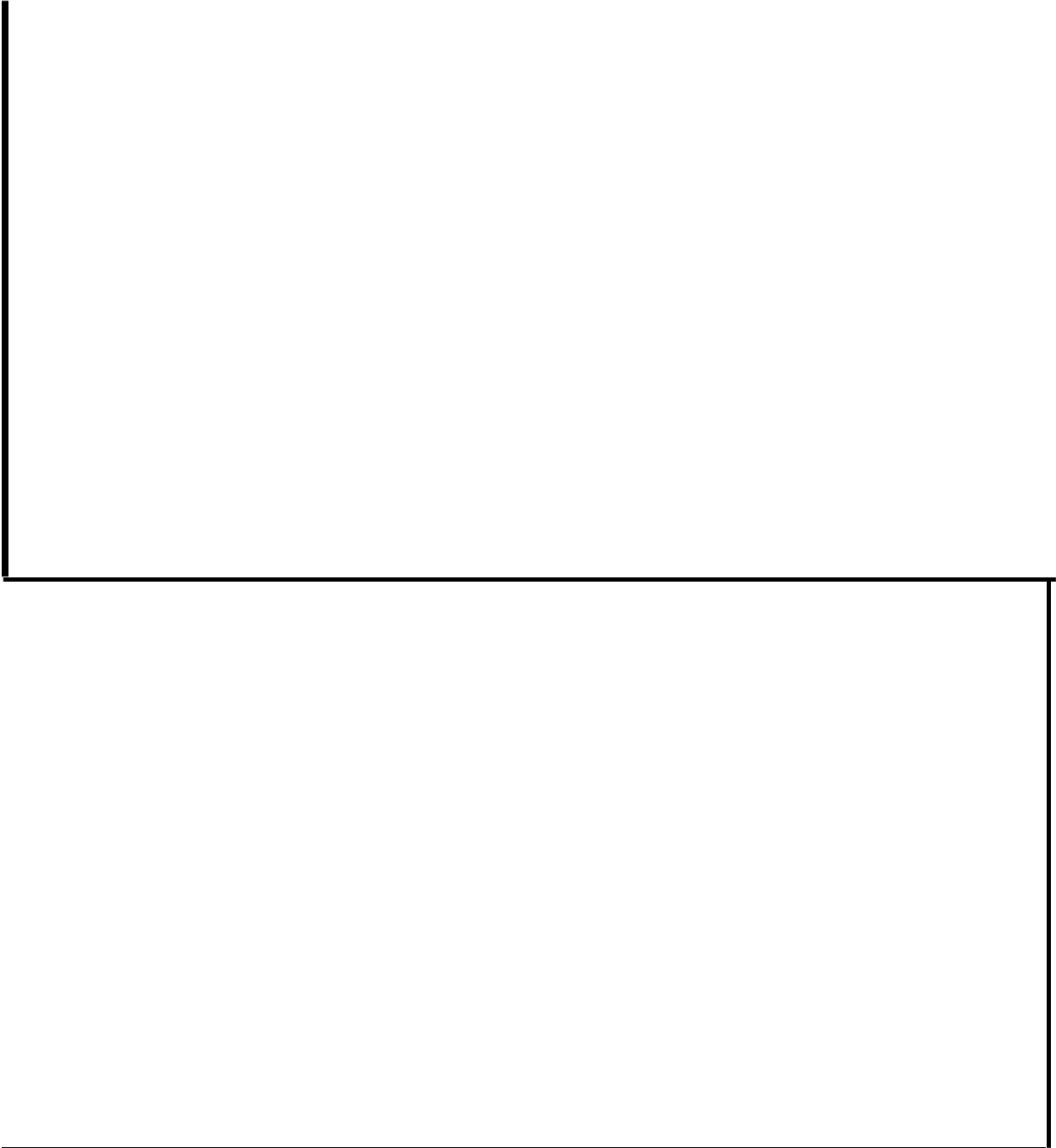
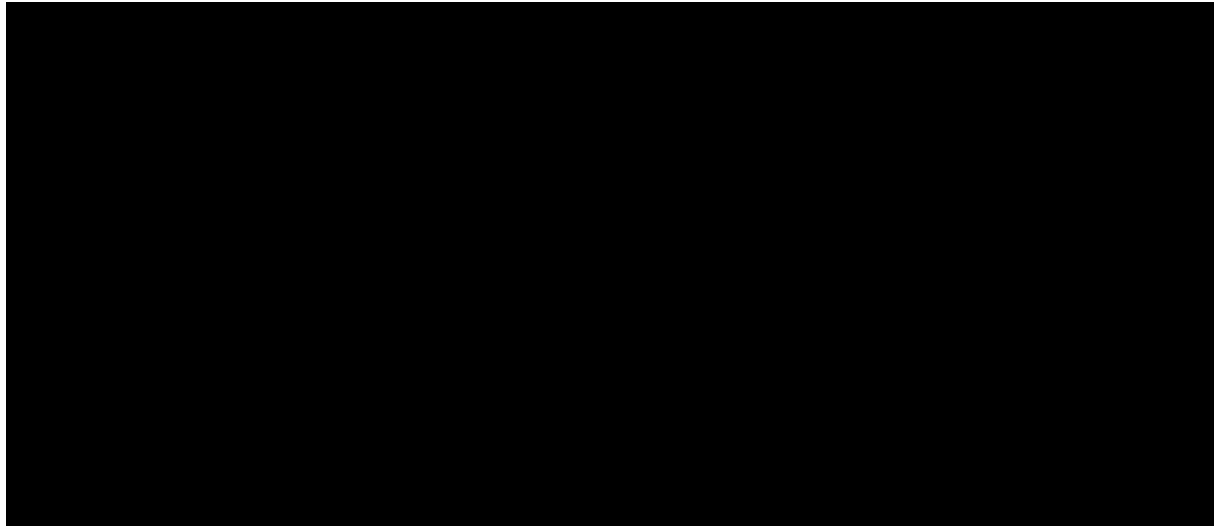


Table of Contents

CRITICAL SKILL 10.1: Inheritance Fundamentals	2
---	---



CRITICAL SKILL 10.2: Base Class Access Control	7
CRITICAL SKILL 10.3: Using protected Members	9
CRITICAL SKILL 10.4: Calling Base Class Constructors	14
CRITICAL SKILL 10.5: Creating a Multilevel Hierarchy	22
CRITICAL SKILL 10.6: Inheriting Multiple Base Classes	25
CRITICAL SKILL 10.7: When Constructor and Destructor Functions Are Executed	26
CRITICAL SKILL 10.8: Pointers to Derived Types	27
CRITICAL SKILL 10.9: Virtual Functions and Polymorphism	28
CRITICAL SKILL 10.10: Pure Virtual Functions and Abstract Classes	37

This module discusses three features of C++ that directly relate to object-oriented programming: inheritance, virtual functions, and polymorphism. Inheritance is the feature that allows one class to inherit the characteristics of another. Using inheritance, you can create a general class that defines traits common to a set of related items. This class can then be inherited by

other, more specific classes, each adding those things that are unique to it. Built on the foundation of inheritance is the virtual function. The virtual function supports polymorphism, the “one interface, multiple methods” philosophy of object-oriented programming.

1	

CRITICAL SKILL 10.1: Inheritance Fundamentals

In the language of C++, a class that is inherited is called a base class. The class that does the inheriting is called a derived class. Therefore, a derived class is a specialized version of a base class. A derived class inherits all of the members defined by the base class and adds its own, unique elements.

C++ implements inheritance by allowing one class to incorporate another class into its declaration. This is done by specifying a base class when a derived class is declared. Let's begin with a short example that illustrates several of the key features of inheritance. The following program creates a base class called TwoDShape that stores the width and height of a two-dimensional object, and a derived class called Triangle. Pay close attention to the way that Triangle is declared.

```

// A simple class hierarchy.

#include <iostream>
#include <cstring>
using namespace std;

// A class for two-dimensional objects.
class TwoDShape {
public:
    double width;
    double height;

    void showDim() {
        cout << "Width and height are " <<
            width << " and " << height << "\n";
    }
};

// Triangle is derived from TwoDShape.
class Triangle : public TwoDShape { ← Triangle inherits TwoDShape.
public:                                     Notice the syntax.
    char style[20];

    double area() {
        return width * height / 2; ← Triangle can refer to the
    }                                   members of TwoDShape as
                                        if they were part of Triangle.

    void showStyle() {
        cout << "Triangle is " << style << "\n";
    }
};

```

2	

```

int main() {
    Triangle t1;
    Triangle t2;

    t1.width = 4.0;
    t1.height = 4.0;
    strcpy(t1.style, "isosceles");

    t2.width = 8.0;
    t2.height = 12.0;
    strcpy(t2.style, "right");

    cout << "Info for t1:\n";
    t1.showStyle();
    t1.showDim();
    cout << "Area is " << t1.area() << "\n";

    cout << "\n";
    cout << "Info for t2:\n";
    t2.showStyle();
    t2.showDim();
    cout << "Area is " << t2.area() << "\n";

    return 0;
}

```

← All members of **Triangle** are available to **Triangle** objects, even those inherited from **TwoDShape**.

The output from this program is shown here:

```

Info for t1:
Triangle is isosceles
Width and height are 4 and 4
Area is 8

Info for t2:
Triangle is right
Width and height are 8 and 12
Area is 48

```

Here, TwoDShape defines the attributes of a “generic” two-dimensional shape, such as a square, rectangle, triangle, and so on. The Triangle class creates a specific type of TwoDShape, in this case, a triangle. The Triangle class includes all of TwoDShape and adds the field style, the function area(), and the function showStyle(). A description of the type of triangle is stored in style, area() computes and returns the area of the triangle, and showStyle() displays the triangle style.

The following line shows how Triangle inherits TwoDShape:

```
class Triangle : public TwoDShape {
```

Here, TwoDShape is a base class that is inherited by Triangle, which is a derived class. As this example shows, the syntax for inheriting a class is remarkably simple and easy-to-use.

3	

Because Triangle includes all of the members of its base class, TwoDShape, it can access width and height inside area(). Also, inside main(), objects t1 and t2 can refer to width and height directly, as if they were part of Triangle. Figure 10-1 depicts conceptually how TwoDShape is incorporated into Triangle.

One other point: Even though TwoDShape is a base for Triangle, it is also a completely independent, stand-alone class. Being a base class for a derived class does not mean that the base class cannot be used by itself.

The general form for inheritance is shown here:

```
class derived-class : access base-class { // body of derived class }
```

Here, access is optional. However, if present, it must be public, private, or protected. You will learn more about these options later in this module. For now, all inherited classes will use public. Using public means that all the public members of the base class will also be public members of the derived class.

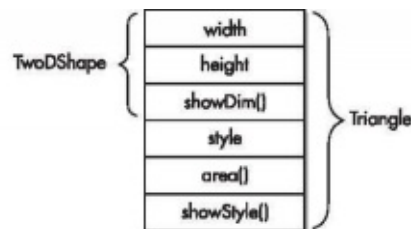


Figure 10-1 A conceptual depiction of the Triangle class

A major advantage of inheritance is that once you have created a base class that defines the attributes common to a set of objects, it can be used to create any number of more specific derived classes. Each derived class can precisely tailor its own classification. For example, here is another class derived from TwoDShape that encapsulates rectangles:


```
// A derived class of TwoDShape for rectangles.
class Rectangle : public TwoDShape {
public:
    bool isSquare() {
        if(width == height) return true;
        return false;
    }

    double area() {
        return width * height;
    }
};
```

The Rectangle class includes TwoDShape and adds the functions isSquare(), which determines if the rectangle is square, and area(), which computes the area of a rectangle.

4	

Member Access and Inheritance

As you learned in Module 8, members of a class are often declared as private to prevent their unauthorized use or tampering. Inheriting a class does not overrule the private access restriction. Thus, even though a derived class includes all of the members of its base class, it cannot access those members of the base class that are private. For example, if width and height are made private in TwoDShape, as shown here, then Triangle will not be able to access them.

```
// Access to private members is not granted to derived classes.

class TwoDShape {
    // these are now private
    double width; ← width and height are now private.
    double height;
public:
    void showDim() {
        cout << "Width and height are " <<
            width << " and " << height << "\n";
    }
};
// Triangle is derived from TwoDShape.
class Triangle : public TwoDShape {
public:
    char style[20];

    double area() {
        return width * height / 2; // Error! Can't access. ←
    }

    void showStyle() {
        cout << "Triangle is " << style << "\n";
    }
};
```

Can't access private members of a base class.

The Triangle class will not compile because the reference to width and height inside the area() function causes an access violation. Since width and height are now private, they are accessible only by other members of their own class. Derived classes have no access to them.

At first, you might think that it is a serious restriction that derived classes do not have access to the private members of base classes, because it would prevent the use of private members in many situations. Fortunately, this is not the case, because C++ provides various solutions. One is to use protected members, which is described in the next section. A second is to use public functions to provide access to private data. As you have seen in the preceding modules, C++ programmers typically grant access to the private members of a class through functions. Functions that provide access to private data are called accessor functions. Here is a rewrite of the TwoDShape class that adds accessor functions for width and height:

5	

```

// Access private data through accessor functions.

#include <iostream>
#include <cstring>
using namespace std;

// A class for two-dimensional objects.
class TwoDShape {
    // these are private
    double width;
    double height;
public:

    void showDim() {
        cout << "Width and height are " <<
            width << " and " << height << "\n";
    }

    // accessor functions
    double getWidth() { return width; }
    double getHeight() { return height; }
    void setWidth(double w) { width = w; }
    void setHeight(double h) { height = h; }
};

// Triangle is derived from TwoDShape.
class Triangle : public TwoDShape {
public:
    char style[20];

    double area() {
        return getWidth() * getHeight() / 2;
    }

    void showStyle() {
        cout << "Triangle is " << style << "\n";
    }
};

int main() {
    Triangle t1;
    Triangle t2;

    t1.setWidth(4.0);
    t1.setHeight(4.0);
    strcpy(t1.style, "isosceles");

    t2.setWidth(8.0);
    t2.setHeight(12.0);
    strcpy(t2.style, "right");
}

```

The accessor functions for width and height

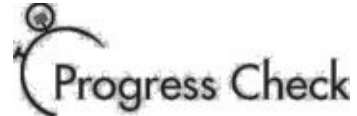
Use the accessor functions to obtain the width and height.

6	

```
cout << "Info for t1:\n";
t1.showStyle();
t1.showDim();
cout << "Area is " << t1.area() << "\n";

cout << "\n";
cout << "Info for t2:\n";
t2.showStyle();
t2.showDim();
cout << "Area is " << t2.area() << "\n";

return 0;
}
```



How is a base class inherited by a derived class?

Does a derived class include the members of its base class?

Does a derived class have access to the

private members of its base class?

CRITICAL SKILL 10.2: Base Class Access Control

As explained, when one class inherits another, the members of the base class become members of the derived class. However, the accessibility of the base class members inside the derived class is determined by the access specifier used when inheriting the base class. The base class access specifier must be public, private, or protected. If the access specifier is not used, then it is private by default if the derived class is a class. If the derived class is a struct, then public is the default. Let's examine the ramifications of using public or private access. (The protected specifier is described in the next section.)

Ask the Expert

Q: I have heard the terms superclass and subclass used in discussions of Java programming. Do these terms have meaning in C++?

A: What Java calls a superclass, C++ calls a base class. What Java calls a subclass, C++ calls a derived class. You will commonly hear both sets of terms applied to a class of either language, but this book will

7	
---	--

continue to use the standard C++ terms. By the way, C# also uses the base class, derived class terminology.

When a base class is inherited as public, all public members of the base class become public members of the derived class. In all cases, the private elements of the base class remain private to that class and are not accessible by members of the derived class. For example, in the following program, the public members of B become public members of D. Thus, they are accessible by other parts of the program.

```
// Demonstrate public inheritance.

#include <iostream>
using namespace std;

class B {
    int i, j;
public:
    void set(int a, int b) { i = a; j = b; }
    void show() { cout << i << " " << j << "\n"; }
};

class D : public B { ←———— Here, B is inherited as public.
    int k;
public:
    D(int x) { k = x; }
    void showk() { cout << k << "\n"; }

    // i = 10; // Error! i is private to B and access is not allowed. ←
};

int main()
{
    D ob(3);

    ob.set(1, 2); // access member of base class
    ob.show();    // access member of base class

    ob.showk();   // uses member of derived class

    return 0;
}
```

Can't access i because it is private to B.

Since `set()` and `show()` are public in B, they can be called on an object of type D from within `main()`. Because `i` and `j` are specified as private, they remain private to B. This is why the line

```
// i = 10; // Error! i is private to B and access is not allowed.
```

is commented-out. D cannot access a private member of B.

The opposite of public inheritance is private inheritance. When the base class is inherited as private, then all public members of the base class become private members of the derived class. For example,

8	
---	--

the program shown next will not compile, because both `set()` and `show()` are now private members of `D`, and thus cannot be called from `main()`.

// Use private inheritance. This program won't compile.

```
#include <iostream>
using namespace std;
```

```
class B {
    int i, j;
public:
    void set(int a, int b) { i = a; j = b; }
    void show() { cout << i << " " << j << "\n"; }
};
```

// Public elements of B become private in D.

```
class D : private B {
    int k;
public:
    D(int x) { k = x; }
    void showk() { cout << k << "\n"; }
};
```

Now, inherit B as private.

```
int main()
{
    D ob(3);

    ob.set(1, 2); // Error, can't access set()
    ob.show();    // Error, can't access show()

    return 0;
}
```

Now, `set()` and `show()` are inaccessible through `D`.

To review: when a base class is inherited as private, public members of the base class become private members of the derived class. This means that they are still accessible by members of the derived class, but cannot be accessed by other parts of your program.

CRITICAL SKILL 10.3: Using protected Members

As you know, a private member of a base class is not accessible by a derived class. This would seem to imply that if you wanted a derived class to have access to some member in the base class, it would need to be public. Of course, making the member public also makes it available to all other code, which may not be desirable. Fortunately, this implication is wrong because C++ allows you to create a protected member. A protected member is public within a class hierarchy, but private outside that hierarchy.

A protected member is created by using the protected access modifier. When a member of a class is declared as protected, that member is, with one important exception, private. The exception occurs

9	

when a protected member is inherited. In this case, the protected member of the base class is accessible by the derived class. Therefore, by using protected, you can create class members that are private to their class but that can still be inherited and accessed by a derived class. The protected specifier can also be used with structures.

Consider this sample program:

```
// Demonstrate protected members.

#include <iostream>
using namespace std;

class B {
    protected:
        int i, j; // private to B, but accessible to D
    public:
        void set(int a, int b) { i = a; j = b; }
        void show() { cout << i << " " << j << "\n"; }
};

class D : public B {
    int k;
    public:
        // D may access B's i and j
        void setk() { k = i*j; }
        void showk() { cout << k << "\n"; }
};

int main()
{
    D ob;

    ob.set(2, 3); // OK, set() is public in B
    ob.show();    // OK, show() is public B

    ob.setk();
    ob.showk();

    return 0;
}
```

Here, i and j are protected.

D can access i and j because they are protected, not private.

Here, because B is inherited by D as public and because i and j are declared as protected, D's function setk() can access them. If i and j were declared as private by B, then D would not have access to them, and the program would not compile.

When a base class is inherited as public, protected members of the base class become protected members of the derived class. When a base class is inherited as private, protected members of the base class become private members of the derived class.

The protected access specifier may occur anywhere in a class declaration, although typically it occurs after the (default) private members are declared and before the public members. Thus, the most common full form of a class declaration is

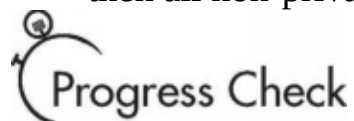
```
class class-name{  
    private members by default protected:  
  
    protected members public:  
  
    public members };
```

Of course, the protected category is optional.

In addition to specifying protected status for members of a class, the keyword `protected` can also act as an access specifier when a base class is inherited. When a base class is inherited as protected, all public and protected members of the base class become protected members of the derived class. For example, in the preceding example, if D inherited B, as shown here:

```
class D : protected B {
```

then all non-private members of B would become protected members of D.



When a base class is inherited as private, public members of the base class become private members of the derived class. True or false?

Can a private member of a base class be made public through inheritance?

To make a member accessible within a hierarchy, but private otherwise, what access specifier do you use?

Ask the Expert

Q:

A:

Can you review public, protected, and private?

When a class member is declared as public, it can be accessed by any other part of a program.

When a member is declared as private, it can be accessed only by members of its class. Further, derived classes do not have access to private base class members. When a member is declared as protected, it can be accessed only by members of its class and by its derived classes. Thus, protected allows a member to be inherited, but to remain private within a class hierarchy.

When a base class is inherited by use of public, its public members become public members of the derived class, and its protected members become protected members of the derived class. When a base class is inherited by use of protected, its public and protected members become protected members of

the derived class. When a base class is inherited by use of private, its public and protected members become private members of the derived class. In all cases, private members of a base class remain private to that base class.

Constructors and Inheritance

In a hierarchy, it is possible for both base classes and derived classes to have their own constructors. This raises an important question: what constructor is responsible for building an object of the derived class, the one in the base class, the one in the derived class, or both? The answer is this: the constructor for the base class constructs the base class portion of the object, and the constructor for the derived class constructs the derived class part. This makes sense because the base class has no knowledge of or access to any element in a derived class. Thus, their construction must be separate. The preceding examples have relied upon the default constructors created automatically by C++, so this was not an issue. However, in practice, most classes will define constructors. Here you will see how to handle this situation.

When only the derived class defines a constructor, the process is straightforward: simply construct the derived class object. The base class portion of the object is constructed automatically using its default constructor. For example, here is a reworked version of Triangle that defines a constructor. It also makes style private since it is now set by the constructor.

12	

```

// Add a constructor to Triangle.

#include <iostream>
#include <cstring>
using namespace std;

// A class for two-dimensional objects.
class TwoDShape {
    // these are private
    double width;
    double height;
public:
    void showDim() {
        cout << "Width and height are " <<
            width << " and " << height << "\n";
    }

    // accessor functions
    double getWidth() { return width; }
    double getHeight() { return height; }
    void setWidth(double w) { width = w; }
    void setHeight(double h) { height = h; }
};

// Triangle is derived from TwoDShape.
class Triangle : public TwoDShape {
    char style[20]; // now private
public:

    // Constructor for Triangle.
    Triangle(char *str, double w, double h) {
        // Initialize the base class portion.
        setWidth(w);
        setHeight(h);
        // Initialize the derived class portion.
        strcpy(style, str);
    }

    double area() {
        return getWidth() * getHeight() / 2;
    }

    void showStyle() {
        cout << "Triangle is " << style << "\n";
    }
};

```

Initialize the TwoDShape portion of Triangle.

Initialize style, which is specific to Triangle.

Here, Triangle's constructor initializes the members of TwoDShape that it inherits along with its own style field.

13	

When both the base class and the derived class define constructors, the process is a bit more complicated, because both the base class and derived class constructors must be executed.

CRITICAL SKILL 10.4: Calling Base Class Constructors

When a base class has a constructor, the derived class must explicitly call it to initialize the base class portion of the object. A derived class can call a constructor defined by its base class by using an expanded form of the derived class' constructor declaration. The general form of this expanded declaration is shown here:

```
derived-constructor(arg-list) : base-cons(arg-list); {  
  
body of derived constructor  
  
}
```

Here, base-cons is the name of the base class inherited by the derived class. Notice that a colon separates the constructor declaration of the derived class from the base class constructor. (If a class inherits more than one base class, then the base class constructors are separated from each other by commas.)

The following program shows how to pass arguments to a base class constructor. It defines a constructor for TwoDShape that initializes the width and height properties.

14	


```

// Add a constructor to TwoDShape.

#include <iostream>
#include <cstring>
using namespace std;

// A class for two-dimensional objects.
class TwoDShape {
    // these are private
    double width;
    double height;
public:

    // Constructor for TwoDShape.
    TwoDShape(double w, double h) {
        width = w;
        height = h;
    }

    void showDim() {
        cout << "Width and height are " <<
            width << " and " << height << "\n";
    }

    // accessor functions
    double getWidth() { return width; }
    double getHeight() { return height; }
    void setWidth(double w) { width = w; }
    void setHeight(double h) { height = h; }
};

// Triangle is derived from TwoDShape.
class Triangle : public TwoDShape {
    char style[20]; // now private
public:

    // Constructor for Triangle.
    Triangle(char *str, double w,
        double h) : TwoDShape(w, h) { ← Call the TwoDShape constructor
        strcpy(style, str);
    }

    double area() {
        return getWidth() * getHeight() / 2;
    }

    void showStyle() {

```


15	

```

        cout << "Triangle is: " << style << "\n";
    }
}

int main() {
    Triangle t1("isosceles", 4.0, 4.0);
    Triangle t2("right", 8.0, 12.0);

    cout << "Info for t1:\n";
    t1.showStyle();
    t1.showDim();
    cout << "Area is: " << t1.area() << "\n";

    cout << "\n";
    cout << "Info for t2:\n";
    t2.showStyle();
    t2.showDim();
    cout << "Area is: " << t2.area() << "\n";

    return 0;
}

```

Here, Triangle() calls TwoDShape with the parameters w and h, which initializes width and height using these values. Triangle no longer initializes these values itself. It need only initialize the value unique to it: style. This leaves TwoDShape free to construct its subobject in any manner that it so

chooses. Furthermore, TwoDShape can add functionality about which existing derived classes have no knowledge, thus preventing existing code from breaking.

Any form of constructor defined by the base class can be called by the derived class' constructor. The constructor executed will be the one that matches the arguments. For example, here are expanded versions of both TwoDShape and Triangle that include additional constructors:

```
// Add a constructor to TwoDShape.

#include <iostream>
#include <cstring>
using namespace std;

// A class for two-dimensional objects.
class TwoDShape {
    // these are private
    double width;
    double height;
public:
```

```

// Default constructor,
TwoDShape() {
    width = height = 0.0;
}

// Constructor for TwoDShape,
TwoDShape(double w, double h) {
    width = w;
    height = h;
}

// Construct object with equal width and height.
TwoDShape(double x) {
    width = height = x;
}

void showDim() {
    cout << "Width and height are " <<
        width << " and " << height << "\n";
}

// accessor functions
double getWidth() { return width; }
double getHeight() { return height; }
void setWidth(double w) { width = w; }
void setHeight(double h) { height = h; }
};

// Triangle is derived from TwoDShape.
class Triangle : public TwoDShape {
    char style[20]; // now private.
public:

    /* A default constructor. This automatically invokes
       the default constructor of TwoDShape. */
    Triangle() {
        strcpy(style, "unknown");
    }

    // Constructor with three parameters.
    Triangle(char *str, double w,
              double h) : TwoDShape(w, h) {
        strcpy(style, str);
    }

    // Construct an isosceles triangle.
    Triangle(double x) : TwoDShape(x) {
        strcpy(style, "isosceles");
    }

```

Various TwoDShape constructors

Various Triangle constructors

17	


```

    }

    double area() {
        return getWidth() * getHeight() / 2;
    }

    void showStyle() {
        cout << "Triangle is " << style << "\n";
    }
};

int main() {
    Triangle t1;
    Triangle t2("right", 8, 0, 12, 0);
    Triangle t3(4, 0);

    t1 = t2;

    cout << "Info for t1: \n";
    t1.showStyle();
    t1.showDim();
    cout << "Area is " << t1.area() << "\n";

    cout << "\n";

    cout << "Info for t2: \n";
    t2.showStyle();
    t2.showDim();
    cout << "Area is " << t2.area() << "\n";

    cout << "\n";

    cout << "Info for t3: \n";
    t3.showStyle();
    t3.showDim();
    cout << "Area is " << t3.area() << "\n";

    cout << "\n";

    return 0;
}

```

Here is the output from this version:

Info for t1:

Triangle is right

Width and height are 8 and 12

Area is 48

Info for t2: Triangle is right Width and height are 8 and 12 Area is 48

Info for t3: Triangle is isosceles Width and height are 4 and 4

18	

Area is 8



How does a derived class execute its base class' constructor?

Can parameters be passed to a base class constructor?

What constructor is responsible for initializing the base class portion of a derived object, the one defined by the derived class or the one defined by the base class?

Project 10-1 Extending the Vehicle Class

This project creates a subclass of the Vehicle class first developed in Module 8.

As you should recall, Vehicle encapsulates information about vehicles, including the number of passengers they can carry, their fuel capacity, and their fuel consumption rate. We can use the Vehicle class as a starting point from which more specialized classes are developed. For example, one type of vehicle is a truck. An important attribute of a truck is its cargo capacity. Thus, to create a Truck class, you can inherit Vehicle, adding an instance variable that stores the carrying capacity. In this project, you will create the Truck class. In the process, the instance variables in Vehicle will be made private, and accessor functions are provided to get their values.

Step by Step

Create a file called TruckDemo.cpp, and copy the last implementation of Vehicle from Module 8 into the file.

Create the Truck class, as shown here:

```
// Use Vehicle to create a Truck specialization.
class Truck : public Vehicle {
    int cargocap; // cargo capacity in pounds
public:

    // This is a constructor for Truck.
    Truck(int p, int f,
          int m, int c) : Vehicle(p, f, m)
    {
        cargocap = c;
    }

    // Accessor function for cargocap.
    int get_cargocap() { return cargocap; }
};
```

Here, Truck inherits Vehicle, adding the cargocap member. Thus, Truck includes all of the general vehicle attributes defined by Vehicle. It need add only those items that are unique to its own class.

3. Here is an entire program that demonstrates the Truck class:

```
// Create a subclass of Vehicle called Truck.

#include <iostream>
using namespace std;

// Declare the Vehicle class.
class Vehicle {
    // These are private.
    int passengers; // number of passengers
    int fuelcap;    // fuel capacity in gallons
    int mpg;        // fuel consumption in miles per gallon
public:
    // This is a constructor for Vehicle.
    Vehicle(int p, int f, int m) {
        passengers = p;
        fuelcap = f;
        mpg = m;
    }

    // Compute and return the range.
    int range() { return mpg * fuelcap; }

    // Accessor functions.
    int get_passengers() { return passengers; }
    int get_fuelcap() { return fuelcap; }
    int get_mpg() { return mpg; }
};

// Use Vehicle to create a Truck specialization
class Truck : public Vehicle {
    int cargocap; // cargo capacity in pounds
public:
    // This is a constructor for Truck.
    Truck(int p, int f,
          int m, int c) : Vehicle(p, f, m
    {
        cargocap = c;
    }

    // Accessor function for cargocap.
    int get_cargocap() { return cargocap; }
};
```


20	


```

int main() {

    // construct some trucks
    Truck semi(2, 200, 7, 44000);
    Truck pickup(3, 28, 15, 2000);
    int dist = 252;

    cout << "Semi can carry " << semi.get_cargocap() <<
           " pounds.\n";
    cout << "It has a range of " <<
           semi.range() << " miles.\n";
    cout << "To go " << dist << " miles semi needs " <<
           dist / semi.get_mpg() <<
           " gallons of fuel.\n\n";

    cout << "Pickup can carry " << pickup.get_cargocap() <<
           " pounds.\n";
    cout << "It has a range of " <<
           pickup.range() << " miles.\n";

    cout << "To go " << dist << " miles pickup needs " <<
           dist / pickup.get_mpg() <<
           " gallons of fuel.\n";

    return 0;
}

```

4. The output from this program is shown here:

```
Semi can carry 44000 pounds.  
It has a range of 1400 miles.  
To go 252 miles semi needs 36 gallons of fuel.  
  
Pickup can carry 2000 pounds.  
It has a range of 420 miles.  
To go 252 miles pickup needs 16 gallons of fuel.
```

Many other types of classes can be derived from Vehicle. For example, the following skeleton creates an off-road class that stores the ground clearance of the vehicle:

```
// Create an off-road vehicle class  
class OffRoad : public Vehicle {  
    int groundClearance; // ground clearance in inches  
public:  
    // ...  
};
```

The key point is that once you have created a base class that defines the general aspects of an object, that base class can be inherited to form specialized classes. Each derived class simply adds its own, unique attributes. This is the essence of inheritance.



CRITICAL SKILL 10.5: Creating a Multilevel Hierarchy

Up to this point, we have been using simple class hierarchies consisting of only a base class and a derived class. However, you can build hierarchies that contain as many layers of inheritance as you like. As mentioned, it is perfectly acceptable to use a derived class as a base class of another. For example, given three classes called A, B, and C, C can be derived from B, which can be derived from A. When this type of situation occurs, each derived class inherits all of the traits found in all of its base classes. In this case, C inherits all aspects of B and A.

To see how a multilevel hierarchy can be useful, consider the following program. In it, the derived class Triangle is used as a base class to create the derived class called ColorTriangle.

ColorTriangle inherits all of the traits of Triangle and TwoDShape, and adds a field called color, which holds the color of the triangle.

```

// A multilevel hierarchy.

#include <iostream>
#include <cstring>
using namespace std;

// A class for two-dimensional objects.
class TwoDShape {
    // these are private
    double width;
    double height;
public:

    // Default constructor.
    TwoDShape() {
        width = height = 0.0;
    }

    // Constructor for TwoDShape.
    TwoDShape(double w, double h) {
        width = w;
        height = h;
    }

    // Construct object with equal width and height.
    TwoDShape(double x) {
        width = height = x;
    }

    void showDim() {
        cout << "Width and height are " <<
            width << " and " << height << "\n";
    }
}

```

22	

```

// accessor functions
double getWidth() { return width; }
double getHeight() { return height; }
void setWidth(double w) { width = w; }
void setHeight(double h) { height = h; }
};

// Triangle is derived from TwoDShape.
class Triangle : public TwoDShape {
    char style[20]; // now private
public:

    /* A default constructor. This automatically invokes
       the default constructor of TwoDShape. */
    Triangle() {
        strcpy(style, "unknown");
    }

    // Constructor with three parameters.
    Triangle(char *str, double w,
             double h) : TwoDShape(w, h) {
        strcpy(style, str);
    }

    // Construct an isosceles triangle.
    Triangle(double x) : TwoDShape(x) {
        strcpy(style, "isosceles");
    }

    double area() {
        return getWidth() * getHeight() / 2;
    }

    void showStyle() {
        cout << "Triangle is " << style << "\n";
    }
};

```



```

// Extend Triangle.
class ColorTriangle : public Triangle { ←———— ColorTriangle inherits Triangle,
    char color[20];                      which inherits TwoDShape.
public:
    ColorTriangle(char *clr, char *style, double w,
                  double h) : Triangle(style, w, h) {
        strcpy(color, clr);
    }

    // Display the color.
    void showColor() {
        cout << "Color is " << color << "\n";
    }
};

int main() {
    ColorTriangle t1("Blue", "right", 8.0, 12.0);
    ColorTriangle t2("Red", "isosceles", 2.0, 2.0);

    cout << "Info for t1:\n";
    t1.showStyle();
    t1.showDim();
    t1.showColor();
    cout << "Area is " << t1.area() << "\n";

    cout << "\n";

    cout << "Info for t2:\n";
    t2.showStyle();
    t2.showDim();
    t2.showColor();
    cout << "Area is " << t2.area() << "\n";

    return 0;
}

```

A ColorTriangle object can call functions defined by itself and its base classes.

The output of this program is shown here:

```
Info for t1:  
Triangle is right  
Width and height are 8 and 12  
Color is Blue  
Area is 48  
  
Info for t2:  
Triangle is isosceles  
Width and height are 2 and 2  
Color is Red  
Area is 2
```

Because of inheritance, ColorTriangle can make use of the previously defined classes of Triangle and TwoDShape, adding only the extra information it needs for its own, specific application. This is part of the value of inheritance; it allows the reuse of code.

24	

This example illustrates one other important point. In a class hierarchy, if a base class constructor requires parameters, then all derived classes must pass those parameters “up the line.” This is true whether or not a derived class needs parameters of its own.

CRITICAL SKILL 10.6: Inheriting Multiple Base Classes

In C++, it is possible for a derived class to inherit two or more base classes at the same time. For example, in this short program, D inherits both B1 and B2:

```
// An example of multiple base classes.

#include <iostream>
using namespace std;

class B1 {
protected:
    int x;
public:
    void showx() { cout << x << "\n"; }
};

class B2 {
protected:
    int y;
public:
    void showy() { cout << y << "\n"; }
};

// Inherit multiple base classes.
class D: public B1, public B2 { ← Here, D inherits both B1
public:                                     and B2 at the same time.
    /* x and y are accessible because they are
       protected in B1 and B2, not private. */
    void set(int i, int j) { x = i; y = j; }
};

int main()
{
    D ob;

    ob.set(10, 20); // provided by D
    ob.showx();     // from B1
    ob.showy();     // from B2

    return 0;
}
```

As this example illustrates, to cause more than one base class to be inherited, you must use a comma-separated list. Further, be sure to use an access specifier for each base class inherited.

25	

CRITICAL SKILL 10.7: When Constructor and Destructor Functions Are Executed

Because a base class, a derived class, or both can contain constructors and/or destructors, it is important to understand the order in which they are executed. Specifically, when an object of a derived class comes into existence, in what order are the constructors called? When the object goes out of existence, in what order are the destructors called? To answer these questions, let's begin with this simple program:

```
#include <iostream>
using namespace std;

class B {
public:
    B() { cout << "Constructing base portion\n"; }
    ~B() { cout << "Destructing base portion\n"; }
};

class D: public B {
public:
    D() { cout << "Constructing derived portion\n"; }
    ~D() { cout << "Destructing derived portion\n"; }
};

int main()
{
    D ob;

    // do nothing but construct and destruct ob

    return 0;
}
```

As the comment in `main()` indicates, this program simply constructs and then destroys an object called `ob`, which is of class `D`. When executed, this program displays

Constructing base portion Constructing derived portion Destructing derived portion Destructing base portion

As the output shows, first the constructor for `B` is executed, followed by the constructor of `D`. Next (since `ob` is immediately destroyed in this program), the destructor of `D` is called, followed by that of `B`.

The results of the foregoing experiment can be generalized as follows: When an object of a derived class is created, the base class constructor is called first, followed by the constructor for the derived class. When a derived object is destroyed, its destructor is called first, followed by that of the base class. Put differently, constructors are executed in the order of their derivation. Destructors are executed in reverse order of derivation. In the case of a multilevel class hierarchy (that is, where a derived class becomes the base class for another derived class), the same general rule applies: Constructors are called

in order of derivation; destructors are called in reverse order. When a class inherits more than one base class at a time, constructors are called in order from left to right as specified in the derived class' inheritance list. Destructors are called in reverse order right to left.



Can a derived class be used as a base class for another derived class?

In a class hierarchy, in what order are the constructors called?

In a class hierarchy, in what order are the destructors called?

Ask the Expert

Q:

A:

Why are constructors called in order of derivation, and destructors called in reverse order?

If you think about it, it makes sense that constructors are executed in order of derivation.

Because a base class has no knowledge of any derived class, any initialization it needs to perform is separate from, and possibly prerequisite to, any initialization performed by the derived class. Therefore, the base

class constructor must be executed first.

Likewise, it is quite sensible that destructors be executed in reverse order of derivation. Since the base class underlies a derived class, the destruction of the base class implies the destruction of the derived class. Therefore, the derived destructor must be called before the object is fully destroyed.

CRITICAL SKILL 10.8: Pointers to Derived Types

Before moving on to virtual functions and polymorphism, it is necessary to discuss an important aspect of pointers. Pointers to base classes and derived classes are related in ways that other types of pointers are not. In general, a pointer of one type cannot point to an object of another type. However, base class pointers and derived objects are the exceptions to this rule. In C++, a base class pointer can also be used to point to an object of any class derived from that base. For example, assume that you have a base class called B and a class called D, which is derived from B. Any pointer declared as a pointer to B can also be used to point to an object of type D. Therefore, given

```
B *p;    // pointer to object of type B
B B_ob;  // object of type B
D D_ob;  // object of type D
```

both of the following statements are perfectly valid:

```
p = &B_ob; // p points to object of type B
p = &D_ob; /* p points to object of type D,
            which is an object derived from B */
```

A base pointer can be used to access only those parts of a derived object that were inherited from the base class. Thus, in this example, `p` can be used to access all elements of `D_ob` inherited from `B_ob`. However, elements specific to `D_ob` cannot be accessed through `p`.

Another point to understand is that although a base pointer can be used to point to a derived object, the reverse is not true. That is, you cannot access an object of the base type by using a derived class pointer.

As you know, a pointer is incremented and decremented relative to its base type. Therefore, when a base class pointer is pointing at a derived object, incrementing or decrementing it will not make it point to the next object of the derived class. Instead, it will point to (what it thinks is) the next object of the base class. Therefore, you should consider it invalid to increment or decrement a base class pointer when it is pointing to a derived object.

The fact that a pointer to a base type can be used to point to any object derived from that base is extremely important, and fundamental to C++. As you will soon learn, this flexibility is crucial to the way C++ implements runtime polymorphism.

References to Derived Types

Similar to the action of pointers just described, a base class reference can be used to refer to an object of a derived type. The most common application of this is found in function parameters. A base class reference parameter can receive objects of the base class as well as any other type derived from that base.

CRITICAL SKILL 10.9: Virtual Functions and Polymorphism

The foundation upon which C++ builds its support for polymorphism consists of inheritance and base class pointers. The specific feature that actually implements polymorphism is the virtual function. The remainder of this module examines this important feature.

Virtual Function Fundamentals

A virtual function is a function that is declared as virtual in a base class and redefined in one or more derived classes. Thus, each derived class can have its own version of a virtual function.

What makes virtual functions interesting is what happens when a base class pointer is used to call one. When a virtual function is called through a base class pointer, C++ determines which version of that function to call based upon the type of the object pointed to by the pointer. This determination is made at runtime. Thus, when different objects are pointed to, different versions of the virtual function are executed. In other words, it is the type of the object being pointed to (not the type of the pointer) that determines which version of the virtual function will be executed. Therefore, if a base class contains a virtual function and if two or more different classes are derived from that base class, then when different types of objects are pointed to through a base class pointer, different versions of the virtual

function are executed. The same effect occurs when a virtual function is called through a base class reference.

You declare a virtual function as `virtual` inside a base class by preceding its declaration with the keyword `virtual`. When a virtual function is redefined by a derived class, the keyword `virtual` need not be repeated (although it is not an error to do so).

A class that includes a virtual function is called a polymorphic class. This term also applies to a class that inherits a base class containing a virtual function.

The following program demonstrates a virtual function:

29	

```

// A short example that uses a virtual function.

#include <iostream>
using namespace std;

class B {
public:
    virtual void who() { // specify a virtual function
        cout << "Base\n";
    }
};

class D1 : public B {
public:
    void who() { // redefine who() for D1
        cout << "First derivation\n";
    }
};

class D2 : public B {
public:
    void who() { // redefine who() for D2
        cout << "Second derivation\n";
    }
};

int main()
{
    B base_obj;
    B *p;
    D1 D1_obj;
    D2 D2_obj;

    p = &base_obj;
    p->who(); // access B's who

    p = &D1_obj;
    p->who(); // access D1's who

    p = &D2_obj;
    p->who(); // access D2's who

    return 0;
}

```

Declare a virtual function.

Redefine the virtual function for D1.

Redefine the virtual function a second time for D2.

Call the virtual function through a base class pointer.

This program produces the following output:

Base

First derivation

Second derivation

Let's examine the program in detail to understand how it works.

As you can see, in B, the function `who()` is declared as `virtual`. This means that the function can be redefined by a derived class. Inside both D1 and D2, `who()` is redefined relative to each class. Inside `main()`, four variables are declared: `base_obj`, which is an object of type B; `p`, which is a pointer to

B

30	

objects; and D1_obj and D2_obj, which are objects of the two derived classes. Next, p is assigned the address of base_obj, and the who() function is called. Since who() is declared as virtual, C++ determines at runtime which version of who() to execute based on the type of object pointed to by p. In this case, p points to an object of type B, so it is the version of who() declared in B that is executed. Next, p is assigned the address of D1_obj. Recall that a base class pointer can refer to an object of any derived class. Now, when who() is called, C++ again checks to see what type of object is pointed to by p and, based on that type, determines which version of who() to call. Since p points to an object of type D1, that version of who() is used. Likewise, when p is assigned the address of D2_obj, the version of who() declared inside D2 is executed.

To review: When a virtual function is called through a base class pointer, the version of the virtual function actually executed is determined at runtime by the type of object being pointed to.

Although virtual functions are normally called through base class pointers, a virtual function can also be called normally, using the standard dot operator syntax. This means that in the preceding example, it would have been syntactically correct to access who() using this statement:

```
D1_obj.who();
```

However, calling a virtual function in this manner ignores its polymorphic attributes. It is only when a virtual function is accessed through a base class pointer (or reference) that runtime polymorphism is achieved.

At first, the redefinition of a virtual function in a derived class seems to be a special form of function overloading. However, this is not the case. In fact, the two processes are fundamentally different. First, an overloaded function must differ in its type and/or number of parameters, while a redefined virtual function must have exactly the same type and number of parameters. In fact, the prototypes for a virtual function and its redefinitions must be exactly the same. If the prototypes differ, then the function is simply considered to be overloaded, and its virtual nature is lost. Another restriction is that a virtual function must be a member, not a friend, of the class for which it is defined. However, a virtual function can be a friend of another class. Also, it is permissible for destructors, but not constructors, to be virtual.

Because of the restrictions and differences between overloading normal functions and redefining virtual functions, the term overriding is used to describe the redefinition of a virtual function.

Virtual Functions Are Inherited

Once a function is declared as virtual, it stays virtual no matter how many layers of derived classes it may pass through. For example, if D2 is derived from D1 instead of B, as shown in the next example, then `who()` is still virtual:

31	

```
// Derive from D1, not B.
class D2 : public D1 {
public:
    void who() { // define who() relative to second_d
        cout << "Second derivation\n";
    }
};
```

When a derived class does not override a virtual function, then the function as defined in the base class is used. For example, try this version of the preceding program. Here, D2 does not override who():

```

#include <iostream>
using namespace std;

class B {
public:
    virtual void who() {
        cout << "Base\n";
    }
};

class D1 : public B {
public:
    void who() {
        cout << "First derivation\n";
    }
};

class D2 : public B {
// who() not defined ← D2 does not override who().
};

int main()
{
    B base_obj;
    B *p;
    D1 D1_obj;
    D2 D2_obj;

    p = &base_obj;
    p->who(); // access B's who()

    p = &D1_obj;
    p->who(); // access D1's who()

    p = &D2_obj;
    p->who(); /* access B's who() because ← This calls the who() defined by B.
               D2 does not redefine it */

    return 0;
}

```

The program now outputs the following:

32	

Base
First derivation
Base

Because D2 does not override `who()`, the version of `who()` defined in B is used instead.

Keep in mind that inherited characteristics of virtual are hierarchical. Therefore, if the preceding example is changed such that D2 is derived from D1 instead of B, then when `who()` is called on an object of type D2, it will not be the `who()` inside B, but the version of `who()` declared inside D1 that is called since it is the class closest to D2.

Why Virtual Functions?

As stated earlier, virtual functions in combination with derived types allow C++ to support runtime polymorphism. Polymorphism is essential to object-oriented programming, because it allows a generalized class to specify those functions that will be common to all derivatives of that class, while allowing a derived class to define the specific implementation of some or all of those functions. Sometimes this idea is expressed as follows: the base class dictates the general interface that any object derived from that class will have, but lets the derived class define the actual method used to implement that interface. This is why the phrase “one interface, multiple methods” is often used to describe polymorphism.

Part of the key to successfully applying polymorphism is understanding that the base and derived classes form a hierarchy, which moves from greater to lesser generalization (base to derived). When designed correctly, the base class provides all of the elements that a derived class can use directly. It also defines those functions that the derived class must implement on its own. This allows the derived class the flexibility to define its own methods, and yet still enforces a consistent interface. That is, since the form of the interface is defined by the base class, any derived class will share that common interface. Thus, the use of virtual functions makes it possible for the base class to define the generic interface that will be used by all derived classes.

At this point, you might be asking yourself why a consistent interface with multiple implementations is important. The answer, again, goes back to the

central driving force behind object-oriented programming: It helps the programmer handle increasingly complex programs. For example, if you develop your program correctly, then you know that all objects you derive from a base class are accessed in the same general way, even if the specific actions vary from one derived class to the next. This means that you need to deal with only one interface, rather than several. Also, your derived class is free to use any or all of the functionality provided by the base class. You need not reinvent those elements.

The separation of interface and implementation also allows the creation of class libraries, which can be provided by a third party. If these libraries are implemented correctly, they will provide a common interface that you can use to derive classes of your own that meet your specific needs. For example, both the Microsoft Foundation Classes (MFC) and the newer .NET Framework Windows Forms class library support Windows programming. By using these classes, your program can inherit much of the

functionality required by a Windows program. You need add only the features unique to your application. This is a major benefit when programming complex systems.

Applying Virtual Functions

To better understand the power of virtual functions, we will apply it to the TwoDShape class. In the preceding examples, each class derived from TwoDShape defines a function called area(). This suggests that it might be better to make area() a virtual function of the TwoDShape class, allowing each derived class to override it, defining how the area is calculated for the type of shape that the class encapsulates. The following program does this. For convenience, it also adds a name field to TwoDShape. (This makes it easier to demonstrate the classes.)

```
// Use virtual functions and polymorphism.

#include <iostream>
#include <cstring>
using namespace std;

// A class for two-dimensional objects.
class TwoDShape {
    // these are private
    double width;
    double height;

    // add a name field
    char name[20];
public:

    // Default constructor.
    TwoDShape() {
        width = height = 0.0;
        strcpy(name, "unknown");
    }

    // Constructor for TwoDShape.
    TwoDShape(double w, double h, char *n) {
        width = w;
        height = h;
        strcpy(name, n);
    }

    // Construct object with equal width and height.
    TwoDShape(double x, char *n) {
        width = height = x;
        strcpy(name, n);
    }
}
```

34	

```

void showDim() {
    cout << "Width and height are " <<
        width << " and " << height << "\n";
}

// accessor functions
double getWidth() { return width; }
double getHeight() { return height; }
void setWidth(double w) { width = w; }
void setHeight(double h) { height = h; }
char *getName() { return name; }

// Add area() to TwoDShape and make it virtual.
virtual double area() {
    cout << "Error: area() must be overridden.\n";
    return 0.0;
}

};

// Triangle is derived from TwoDShape.
class Triangle : public TwoDShape {
    char style[20]; // now private
public:

    /* A default constructor. This automatically invokes
       the default constructor of TwoDShape. */
    Triangle() {
        strcpy(style, "unknown");
    }

    // Constructor with three parameters.
    Triangle(char *str, double w,
             double h) : TwoDShape(w, h, "triangle") {
        strcpy(style, str);
    }

    // Construct an isosceles triangle.
    Triangle(double x) : TwoDShape(x, "triangle") {
        strcpy(style, "isosceles");
    }

    // This now overrides area() declared in TwoDShape.
    double area() {
        return getWidth() * getHeight() / 2;
    }

    void showStyle() {
        cout << "Triangle is " << style << "\n";
    }
};

```

← The `area()` function is now virtual.

← Override `area()` in Triangle.

35	

```

// A derived class of TwoDShape for rectangles.
class Rectangle : public TwoDShape {
public:

    // Construct a rectangle.
    Rectangle(double w, double h) :
        TwoDShape(w, h, "rectangle") { }

    // Construct a square.

    Rectangle(double x) :
        TwoDShape(x, "rectangle") { }

    bool isSquare() {
        if(getWidth() == getHeight()) return true;
        return false;
    }

    // This is another override of area().
    double area() {
        return getWidth() * getHeight();
    }
};

int main() {
    // declare an array of pointers to TwoDShape objects.
    TwoDShape *shapes[5];

    shapes[0] = &Triangle("right", 8.0, 12.0);
    shapes[1] = &Rectangle(10);
    shapes[2] = &Rectangle(10, 4);
    shapes[3] = &Triangle(7.0);
    shapes[4] = &TwoDShape(10, 20, "generic");

    for(int i=0; i < 5; i++) {
        cout << "object is " <<
            shapes[i]->getName() << "\n";

        cout << "Area is " <<
            shapes[i]->area() << "\n";

        cout << "\n";
    }

    return 0;
}

```

← Override `area()` again in `Rectangle`.

← The proper version of `area()` for each object is now called.

The output from the program is shown here:

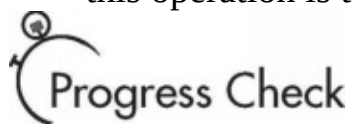
object is triangle Area is 48
object is rectangle Area is 100
object is rectangle

36	

Area is 40
object is triangle Area is 24.5
object is generic
Error: area() must be overridden.
Area is 0

Let's examine this program closely. First, `area()` is declared as virtual in `TwoDShape` class and is overridden by `Triangle` and `Rectangle`. Inside `TwoDShape`, `area()` is given a placeholder implementation that simply informs the user that this function must be overridden by a derived class. Each override of `area()` supplies an implementation that is suitable for the type of object encapsulated by the derived class. Thus, if you were to implement an ellipse class, for example, then `area()` would need to compute the area of an ellipse.

There is one other important feature in the preceding program. Notice in `main()` that `shapes` is declared as an array of pointers to `TwoDShape` objects. However, the elements of this array are assigned pointers to `Triangle`, `Rectangle`, and `TwoDShape` objects. This is valid because a base class pointer can point to a derived class object. The program then cycles through the array, displaying information about each object. Although quite simple, this illustrates the power of both inheritance and virtual functions. The type of object pointed to by a base class pointer is determined at runtime and acted on accordingly. If an object is derived from `TwoDShape`, then its area can be obtained by calling `area()`. The interface to this operation is the same no matter what type of shape is being used.



What is a virtual function?
Why are virtual functions important?

When an overridden virtual function is called through a base class pointer, which version of the function is executed?

CRITICAL SKILL 10.10: Pure Virtual Functions and Abstract Classes

Sometimes you will want to create a base class that defines only a generalized form that will be shared by all of its derived classes, leaving it to each derived class to fill in the details. Such a class determines the nature of the functions that the derived classes must implement, but does not, itself, provide an implementation of one or more of these functions. One way this situation can occur is when a base class is unable to create a meaningful implementation for a function. This is the case with the version of `TwoDShape` used in the preceding example. The definition of `area()` is simply a placeholder. It will not compute and display the area of any type of object.

As you will see as you create your own class libraries, it is not uncommon for a function to have no meaningful definition in the context of its base class. You can handle this situation two ways. One way,

37	

as shown in the previous example, is to simply have it report a warning message. While this approach can be useful in certain situations—such as debugging—it is not usually appropriate. You may have functions that must be overridden by the derived class in order for the derived class to have any meaning. Consider the class `Triangle`. It has no meaning if `area()` is not defined. In this case, you want some way to ensure that a derived class does, indeed, override all necessary functions. The C++ solution to this problem is the pure virtual function.

A pure virtual function is a function declared in a base class that has no definition relative to the base. As a result, any derived class must define its own version—it cannot simply use the version defined in the base. To declare a pure virtual function, use this general form: `virtual type func-name(parameter-list) = 0;`

Here, `type` is the return type of the function, and `func-name` is the name of the function. Using a pure virtual function, you can improve the `TwoDShape` class. Since there is no meaningful concept of area for an undefined two-dimensional figure, the following version of the preceding program declares `area()` as a pure virtual function inside `TwoDShape`. This, of course, means that all classes derived from `TwoDShape` must override `area()`.

```
// Use a pure virtual function.

#include <iostream>
#include <cstring>
using namespace std;

// A class for two-dimensional objects.
class TwoDShape {
    // these are private
    double width;
    double height;

    // add a name field
    char name[20];
public:

    // Default constructor.
    TwoDShape() {
        width = height = 0.0;
        strcpy(name, "unknown");
    }

    // Constructor for TwoDShape.
    TwoDShape(double w, double h, char *n) {
        width = w;
        height = h;
        strcpy(name, n);
    }
}
```

38	

```

// Construct object with equal width and height.
TwoDShape(double x, char *n) {
    width = height = x;
    strcpy(name, n);
}

void showDim() {
    cout << "Width and height are " <<
        width << " and " << height << "\n";
}

// accessor functions
double getWidth() { return width; }
double getHeight() { return height; }
void setWidth(double w) { width = w; }
void setHeight(double h) { height = h; }
char *getName() { return name; }

// area() is now a pure virtual function
virtual double area() = 0;

```

← area() is now a pure virtual function.

```

};

// Triangle is derived from TwoDShape.
class Triangle : public TwoDShape {
    char style[20]; // now private

public:

    /* A default constructor. This automatically invokes
       the default constructor of TwoDShape. */
    Triangle() {
        strcpy(style, "unknown");
    }

    // Constructor with three parameters.
    Triangle(char *str, double w,
             double h) : TwoDShape(w, h, "triangle") {
        strcpy(style, str);
    }

    // Construct an isosceles triangle.
    Triangle(double x) : TwoDShape(x, "triangle") {
        strcpy(style, "isosceles");
    }

    // This now overrides area() declared in TwoDShape.
    double area() {
        return getWidth() * getHeight() / 2;
    }
}

```


39	


```

    void showStyle() {
        cout << "Triangle is " << style << "\n";
    }
};

// A derived class of TwoDShape for rectangles.
class Rectangle : public TwoDShape {
public:

    // Construct a rectangle.
    Rectangle(double w, double h) :
        TwoDShape(w, h, "rectangle") { }

    // Construct a square.
    Rectangle(double x) :
        TwoDShape(x, "rectangle") { }

    bool isSquare() {
        if(getWidth() == getHeight()) return true;
        return false;
    }
    // This is another override of area().
    double area() {
        return getWidth() * getHeight();
    }
};

int main() {
    // declare an array of pointers to TwoDShape objects.
    TwoDShape *shapes[4];

    shapes[0] = &Triangle("right", 8.0, 12.0);
    shapes[1] = &Rectangle(10);
    shapes[2] = &Rectangle(10, 4);
    shapes[3] = &Triangle(7.0);

    for(int i=0; i < 4; i++) {
        cout << "object is " <<
            shapes[i]->getName() << "\n";

        cout << "Area is " <<
            shapes[i]->area() << "\n";

        cout << "\n";
    }

    return 0;
}

```


If a class has at least one pure virtual function, then that class is said to be abstract. An abstract class has one important feature: there can be no objects of that class. To prove this to yourself, try removing the

40	

override of `area()` from the `Triangle` class in the preceding program. You will receive an error when you try to create an instance of `Triangle`. Instead, an abstract class must be used only as a base that other classes will inherit. The reason that an abstract class cannot be used to declare an object is because one or more of its functions have no definition. Because of this, the `shapes` array in the preceding program has been shortened to 4, and a generic `TwoDShape` object is no longer created. As the program illustrates, even if the base class is abstract, you still can use it to declare a pointer of its type, which can be used to point to derived class objects.

Module 10 Mastery Check

A class that is inherited is called a _____ class. The class that does the inheriting is called a _____ class.

Does a base class have access to the members of a derived class? Does a derived class have access to the members of a base class?

Create a derived class of `TwoDShape` called `Circle`. Include an `area()` function that computes the area of the circle.

How do you prevent a derived class from having access to a member of a base class?

Show the general form of a constructor that calls a base class constructor.

Given the following hierarchy:

in what order are the constructors for these classes

called when a Gamma object is instantiated?

How can protected members be accessed?

A base class pointer can refer to a derived class object. Explain why this is important as it relates to function overriding.

What is a pure virtual function? What is an abstract class?

Can an object of an abstract class be instantiated?

Explain how the pure virtual function helps implement the “one interface, multiple methods” aspect of polymorphism.

```
class Alpha { ...  
  
class Beta : public Alpha { ...  
  
Class Gamma : public Beta { ...
```

Module 11

The C++ I/O System

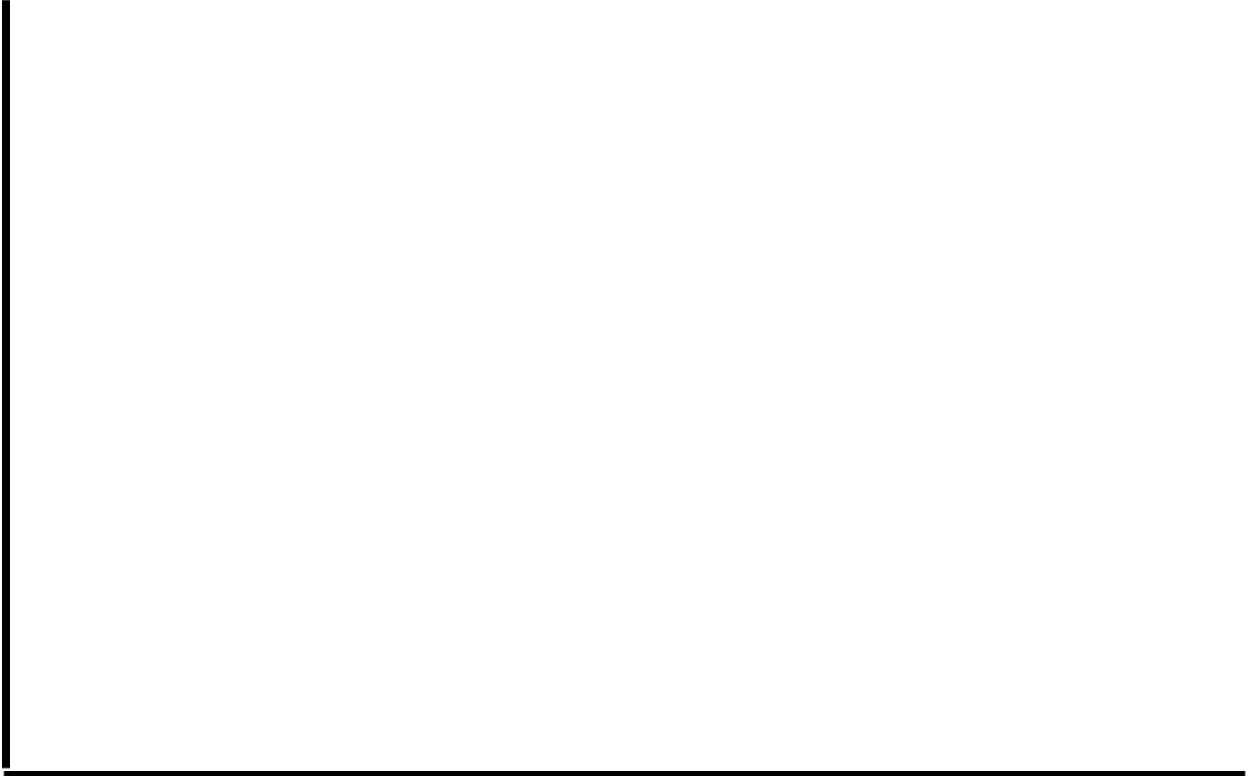


Table of Contents

CRITICAL SKILL 11.1: Understand I/O streams	2
CRITICAL SKILL 11.2: Know the I/O class hierarchy	3
CRITICAL SKILL 11.3: Overload the << and >> operators	4
CRITICAL SKILL 11.4: Format I/O by using ios member functions	10
CRITICAL SKILL 11.5: Format I/O by using manipulators	16
CRITICAL SKILL 11.6: Create your own manipulators	18
CRITICAL SKILL 11.7: Open and close files	20
CRITICAL SKILL 11.8: Read and write text files	23
CRITICAL SKILL 11.9: Read and write binary files	25
CRITICAL SKILL 11.10: Know additional file functions	29

Since the beginning of this book you have been using the C++ I/O system, but you have been doing so without much formal explanation. Since the I/O system is based upon a hierarchy of classes, it was not possible to present its theory and details without first discussing classes and inheritance. Now it is time to examine the C++ I/O system in detail. The C++ I/O system is quite large, and it won't be possible to discuss here every class, function, or feature, but this module will introduce you to the most important and commonly used parts. Specifically, it shows how to overload the `<<` and `>>` operators so that you can input or output objects of classes that you design. It describes how to format output and how to use I/O manipulators. The module ends by discussing file I/O.

Old vs. Modern C++ I/O

There are currently two versions of the C++ object-oriented I/O library in use: the older one that is based upon the original specifications for C++ and the newer one defined by Standard C++. The old I/O library is supported by the header file `<iostream.h>`. The new I/O library is supported by the header

1	
---	--

<iostream>. For the most part, the two libraries appear the same to the programmer. This is because the new I/O library is, in essence, simply an updated and improved version of the old one. In fact, the vast majority of differences between the two occur beneath the surface, in the way that the libraries are implemented—not in how they are used.

From the programmer's perspective, there are two main differences between the old and new C++ I/O libraries. First, the new I/O library contains a few additional features and defines some new data types. Thus, the new I/O library is essentially a superset of the old one. Nearly all programs originally written for the old library will compile without substantive changes when the new library is used. Second, the old-style I/O library was in the global namespace. The new-style library is in the std namespace. (Recall that the std namespace is used by all of the Standard C++ libraries.) Since the old-style I/O library is now obsolete, this book describes only the new I/O library, but most of the information is applicable to the old I/O library as well.

CRITICAL SKILL 11.1: C++ Streams

The most fundamental point to understand about the C++ I/O system is that it operates on streams. A stream is an abstraction that either produces or consumes information. A stream is linked to a physical device by the C++ I/O system. All streams behave in the same manner, even if the actual physical devices they are linked to differ. Because all streams act the same, the same I/O functions and operators can operate on virtually any type of device. For example, the same method that you use to write to the screen can be used to write to a disk or to the printer.

In its most common form, a stream is a logical interface to a file. As C++ defines the term “file,” it can refer to a disk file, the screen, the keyboard, a port, a file on tape, and so on. Although files differ in form and capabilities, all streams are the same. The advantage to this approach is that to you, the programmer, one hardware device will look much like any other. The stream provides a consistent interface.

A stream is linked to a file through an open operation. A stream is disassociated from a file through a close operation.

There are two types of streams: text and binary. A text stream is used with characters. When a text stream is being used, some character translations may take place. For example, when the newline character is output, it may be converted into a carriage return–linefeed sequence. For this reason, there might not be a one-to-one correspondence between what is sent to the stream and what is written to the file. A binary stream can be used with any type of data. No character translations will occur, and there is a one-to-one correspondence between what is sent to the stream and what is actually contained in the file.

One more concept to understand is that of the current location. The current location (also referred to as the current position) is the location in a file where the next file access will occur. For example, if a file is 100 bytes long and half the file has been read, the next read operation will occur at byte 50, which is the current location.

2	

To summarize: In C++, I/O is performed through a logical interface called a stream. All streams have similar properties, and every stream is operated upon by the same I/O functions, no matter what type of file it is associated with. A file is the actual physical entity that contains

The C++ I/O System

the data. Even though files differ, streams do not. (Of course, some devices may not support all operations, such as random-access operations, so their associated streams will not support these operations either.)

The C++ Predefined Streams

C++ contains several predefined streams that are automatically opened when your C++ program begins execution. They are `cin`, `cout`, `cerr`, and `clog`. As you know, `cin` is the stream associated with standard input, and `cout` is the stream associated with standard output. The `cerr` stream is linked to standard output, and so is `clog`. The difference between these two streams is that `clog` is buffered, but `cerr` is not. This means that any output sent to `cerr` is immediately output, but output to `clog` is written only when a buffer is full. Typically, `cerr` and `clog` are streams to which program debugging or error information is written. C++ also opens wide (16-bit) character versions of the standard streams called `wcin`, `wcout`, `wcerr`, and `wclog`. These streams exist to support languages, such as Chinese, that require large character sets. We won't be using them in this book. By default, the C++ standard streams are linked to the console, but they can be redirected to other devices or files by your program. They can also be redirected by the operating system.

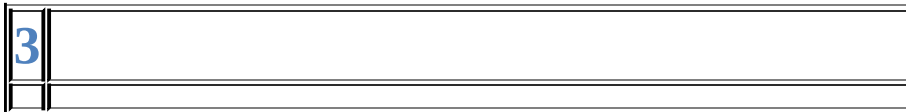
CRITICAL SKILL 11.2: The C++ Stream Classes

As you learned in Module 1, C++ provides support for its I/O system in `<iostream>`. In this header, a rather complicated set of class hierarchies is defined that supports I/O operations. The I/O classes begin with a system of template classes. As you will learn in Module 12, a template defines the form of a class without fully specifying the data upon which it will operate. Once a template class has been defined, specific instances of the template

class can be created. As it relates to the I/O library, Standard C++ creates two specific versions of these template classes: one for 8-bit characters and another for wide characters. These specific versions act like any other classes, and no familiarity with templates is required to fully utilize the C++ I/O system.

The C++ I/O system is built upon two related, but different, template class hierarchies. The first is derived from the low-level I/O class called `basic_streambuf`. This class supplies the basic, low-level input and output operations, and provides the underlying support for the entire C++ I/O system. Unless you are doing advanced I/O programming, you will not need to use `basic_streambuf` directly. The class hierarchy that you will most commonly be working with is derived from `basic_ios`. This is a high-level I/O class that provides formatting, error-checking, and status information related to stream I/O. (A base class for `basic_ios` is called `ios_base`, which defines several traits used by `basic_ios`.) `basic_ios` is used as

base for several derived classes, including `basic_istream`, `basic_ostream`, and `basic_iostream`. These classes are used to create streams capable of input, output, and input/output, respectively.



As explained, the I/O library creates two specific versions of the I/O class hierarchies: one for 8-bit characters and one for wide characters. This book discusses only the 8-bit character classes since they are by far the most frequently used. Here is a list of the mapping of template class names to their character-based versions.

Template Class Name	Equivalent Character-Based Class Name
<code>basic_streambuf</code>	<code>streambuf</code>
<code>basic_ios</code>	<code>ios</code>
<code>basic_istream</code>	<code>istream</code>
<code>basic_ostream</code>	<code>ostream</code>
<code>basic_iostream</code>	<code>iostream</code>
<code>basic_fstream</code>	<code>fstream</code>
<code>basic_ifstream</code>	<code>ifstream</code>
<code>basic_ofstream</code>	<code>ofstream</code>

The character-based names will be used throughout the remainder of this book, since they are the names that you will use in your programs. They are also the same names that were used by the old I/O library. This is why the old and the new I/O library are compatible at the source code level.

One last point: The `ios` class contains many member functions and variables that control or monitor the fundamental operation of a stream. It will be referred to frequently. Just remember that if you include `<iostream>` in your program, you will have access to this important class.



What is a stream? What is a file?

What stream is connected to standard output?

C++ I/O is supported by a sophisticated set of class hierarchies. True or false?

CRITICAL SKILL 11.3: Overloading the I/O Operators

In the preceding modules, when a program needed to output or input the data associated with a class, member functions were created whose only purpose was to output or input the class' data. While there is nothing, in itself, wrong with this approach, C++ allows a much better way of performing I/O operations on classes: by overloading the << and the >> I/O operators.

In the language of C++, the << operator is referred to as the insertion operator because it inserts data into a stream. Likewise, the >> operator is called the extraction operator because it extracts data from a

4	

stream. The operator functions that overload the insertion and extraction operators are generally called inserters and extractors, respectively.

In <iostream>, the insertion and extraction operators are overloaded for all of the C++ built-in types.

Here you will see how to define these operators relative to classes that you create.

Creating Inserters

As a simple first example, let's create an inserter for the version of the ThreeD class shown here:

```
class ThreeD {
public:
    int x, y, z; // 3-D coordinates
    ThreeD(int a, int b, int c) { x = a; y = b; z = c; }
};
```

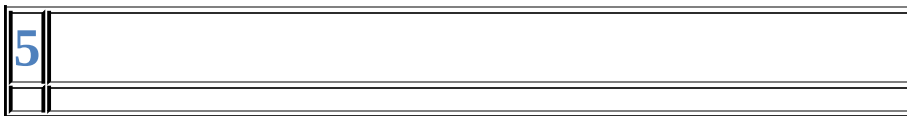
The C++ I/O System

To create an inserter function for an object of type ThreeD, overload the << for it. Here is one way to do this:

```
// Display X, Y, Z coordinates - ThreeD inserter.
ostream &operator<<(ostream &stream, ThreeD obj)
{
    stream << obj.x << ", ";
    stream << obj.y << ", ";
    stream << obj.z << "\n";
    return stream; // return the stream
}
```

Let's look closely at this function, because many of its features are common to all inserter functions. First, notice that it is declared as returning a reference to an object of type ostream. This declaration is necessary so that several inserters of this type can be combined in a compound I/O expression. Next, the function has two parameters. The first is the reference to the stream that occurs on the left side of the << operator. The second parameter is the object that occurs on the right side. (This parameter can also be a reference to the object, if you like.) Inside the function, the three values contained in an object of type ThreeD are output, and stream is returned.

Here is a short program that demonstrates the inserter:




```

// Demonstrate a custom inserter.

#include <iostream>
using namespace std;

class ThreeD {
public:
    int x, y, z; // 3-D coordinates
    ThreeD(int a, int b, int c) { x = a; y = b; z = c; }
};

// Display X, Y, Z coordinates - ThreeD inserter.
ostream &operator<<(ostream &stream, ThreeD obj) ← An inserter for Three D
{
    stream << obj.x << ", ";
    stream << obj.y << ", ";
    stream << obj.z << "\n";
    return stream; // return the stream
}

int main()
{
    ThreeD a(1, 2, 3), b(3, 4, 5), c(5, 6, 7);
    cout << a << b << c; ← Use ThreeD inserter to
                                output coordinates.

    return 0;
}

```

This program displays the following output:

```

1, 2, 3
3, 4, 5
5, 6, 7

```

If you eliminate the code that is specific to the ThreeD class, you are left with the skeleton for an inserter function, as shown here:

```
ostream &operator<<(ostream &stream, class_type obj)
{
    // class specific code goes here
    return stream; // return the stream
}
```

Of course, it is permissible for obj to be passed by reference.

Within wide boundaries, what an inserter function actually does is up to you. However, good programming practice dictates that your inserter should produce reasonable output. Just make sure that you return stream.

Using Friend Functions to Overload Inserters

6	

In the preceding program, the overloaded inserter function is not a member of ThreeD. In fact, neither inserter nor extractor functions can be members of a class. The reason is that when an operator function is a member of a class, the left operand (implicitly passed using the this pointer) is an object of that class. There is no way to change this. However, when inserters are overloaded, the left operand is a stream, and the right operand is an object of the class being output. Therefore, overloaded inserters must be nonmember functions.

The fact that inserters must not be members of the class they are defined to operate on raises a serious question: How can an overloaded inserter access the private elements of a class? In the preceding program, the variables x, y, and z were made public so that the inserter could access them. But hiding data is an important part of OOP, and forcing all data to be public is a serious inconsistency. However, there is a solution: an inserter can be a friend of a class. As a friend of the class for which it is defined, it has access to private data. Here, the ThreeD class and sample program are reworked, with the overloaded inserter declared as a friend:

```
// Use a friend to overload <<.
#include <iostream>
using namespace std;

class ThreeD {
    int x, y, z; // 3-D coordinates - now private
public:
    ThreeD(int a, int b, int c) { x = a; y = b; z = c; }
    friend ostream &operator<<(ostream &stream, ThreeD obj);
};

// Display X, Y, Z coordinates - ThreeD inserter.
ostream &operator<<(ostream &stream, ThreeD obj)
{
    stream << obj.x << ", ";
    stream << obj.y << ", ";
    stream << obj.z << "\n";
    return stream; // return the stream
}

int main()
{
    ThreeD a(1, 2, 3), b(3, 4, 5), c(5, 6, 7);

    cout << a << b << c;

    return 0;
}
```

ThreeD inserter is now a friend and has access to private data.

Notice that the variables `x`, `y`, and `z` are now private to `ThreeD`, but can still be directly accessed by the inserter. Making inserters (and extractors) friends of the classes for which they are defined preserves the encapsulation principle of OOP.

7	
---	--

Overloading Extractors

To overload an extractor, use the same general approach that you use when overloading an inserter. For example, the following extractor inputs 3-D coordinates into an object of type `ThreeD`. Notice that it also prompts the user.

```
// Get three-dimensional values - ThreeD extractor.
istream &operator>>(istream &stream, ThreeD &obj)
{
    cout << "Enter X,Y,Z values: ";
    stream >> obj.x >> obj.y >> obj.z;
    return stream;
}
```

An extractor must return a reference to an object of type `istream`. Also, the first parameter must be a reference to an object of type `istream`. This is the stream that occurs on the left side of the `>>`. The second parameter is a reference to the variable that will be receiving input. Because it is a reference, the second parameter can be modified when information is input.

The skeleton of an extractor is shown here:

```
istream &operator>>(istream &stream, object_type &obj)
{
    // put your extractor code here
    return stream;
}
```

The following program demonstrates the extractor for objects of type `ThreeD`:

8	

```

// Demonstrate a custom extractor.

#include <iostream>
using namespace std;

class ThreeD {
    int x, y, z; // 3-D coordinates
public:
    ThreeD(int a, int b, int c) { x = a; y = b; z = c; }
    friend ostream &operator<<(ostream &stream, ThreeD obj);
    friend istream &operator>>(istream &stream, ThreeD &obj);
};

// Display X, Y, Z coordinates - ThreeD inserter.
ostream &operator<<(ostream &stream, ThreeD obj)
{
    stream << obj.x << ", ";
    stream << obj.y << ", ";
    stream << obj.z << "\n";
    return stream; // return the stream
}

// Get three dimensional values - ThreeD extractor.
istream &operator>>(istream &stream, ThreeD &obj) ← Extractor for ThreeD
{
    cout << "Enter X,Y,Z values: ";
    stream >> obj.x >> obj.y >> obj.z;
    return stream;
}

int main()
{
    ThreeD a(1, 2, 3);

    cout << a;

    cin >> a;
    cout << a;

    return 0;
}

```

A sample run is shown here:

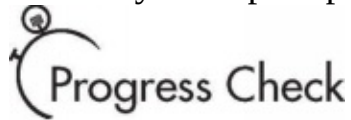
```
1, 2, 3
Enter X, Y, Z values: 5 6 7
5, 6, 7
```


Like inserters, extractor functions cannot be members of the class they are designed to operate upon.

They can be friends or simply independent functions.

9	

Except for the fact that you must return a reference to an object of type `istream`, you can do anything you like inside an extractor function. However, for the sake of structure and clarity, it is best to use extractors only for input operations.



What is an inserter?

adjustfield	basefield	boolalpha	dec
fixed	floatfield	hex	internal
left	oct	right	scientific
showbase	showpoint	showpos	skipws
unitbuf	uppercase		

What is an extractor?

Why are friend functions often used for inserter or extractor functions?

Formatted I/O

Up to this point, the format for inputting or outputting information has been left to the defaults provided by the C++ I/O system. However, you can precisely control the format of your data in either of two ways. The first uses member functions of the `ios` class. The second uses a

special type of function called a manipulator. We will begin by looking at formatting using the `ios` member functions.

CRITICAL SKILL 11.4: Formatting with

the ios Member Functions

Each stream has associated with it a set of format flags that control the way information is formatted by a stream. The ios class declares a bitmask enumeration called `fmtflags` in which the following values are defined. (Technically, these values are defined within `ios_base`, which is a base class for ios.)

10	

These values are used to set or clear the format flags. Some older compilers may not define the `fmtflags` enumeration type. In this case, the format flags will be encoded into a long integer.

When the `skipws` flag is set, leading whitespace characters (spaces, tabs, and newlines) are discarded when performing input on a stream. When `skipws` is cleared, whitespace characters are not discarded.

When the `left` flag is set, output is left-justified. When `right` is set, output is right-justified.

When the `internal` flag is set, a numeric value is padded to fill a field by inserting spaces between any sign or base character. If none of these flags is set, output is right-justified by default.

By default, numeric values are output in decimal. However, it is possible to change the number base. Setting the `oct` flag causes output to be displayed in octal. Setting the `hex` flag causes output to be displayed in hexadecimal. To return output to decimal, set the `dec` flag.

Setting `showbase` causes the base of numeric values to be shown. For example, if the conversion base is hexadecimal, the value `1F` will be displayed as `0x1F`.

By default, when scientific notation is displayed, the `e` is in lowercase. Also, when a hexadecimal value is displayed, the `x` is in lowercase. When uppercase is set, these characters are displayed in uppercase.

Setting `showpos` causes a leading plus sign to be displayed before positive values. Setting `showpoint` causes a decimal point and trailing zeros to be displayed for all floating-point output—whether needed or not.

By setting the `scientific` flag, floating-point numeric values are displayed using scientific notation. When `fixed` is set, floating-point values are displayed using normal notation. When neither flag is set, the compiler chooses an appropriate method.

When `unitbuf` is set, the buffer is flushed after each insertion operation. When `boolalpha` is set, Booleans can be input or output using the keywords `true` and `false`.

Since it is common to refer to the oct, dec, and hex fields, they can be collectively referred to as basefield. Similarly, the left, right, and internal fields can be referred to as adjustfield.

Finally, the scientific and fixed fields can be referenced as floatfield.

Setting and Clearing Format Flags

To set a flag, use the `setf()` function. This function is a member of `ios`. Its most common form is shown here:

```
fmtflags setf(fmtflags flags);
```

This function returns the previous settings of the format flags and turns on those flags specified by flags.

For example, to turn on the `showbase` flag, you can use this statement:

11	

```
stream.setf(ios::showbase);
```

Here, stream is the stream you want to affect. Notice the use of ios:: to qualify showbase. Because showbase is an enumerated constant defined by the ios class, it must be qualified by ios when it is referred to. This principle applies to all of the format flags.

The following program uses setf() to turn on both the showpos and scientific flags:

```
// Use setf().

#include <iostream>
using namespace std;

int main()
{
    // Turn on showpos and scientific flags.
    cout.setf(ios::showpos);
    cout.setf(ios::scientific);
    cout << 123 << " " << 123.23 << " ";

    return 0;
}
```

Set format flag using setf().

The output produced by this program is shown here:

```
+123 +1.232300e+002
```

You can OR together as many flags as you like in a single call. For example, by ORing together scientific and showpos, as shown next, you can change the program so that only one call is made to setf():

```
cout.setf(ios::scientific | ios::showpos);
```

To turn off a flag, use the unsetf() function, whose prototype is shown here: void unsetf(fmtflags flags); The flags specified by flags are cleared. (All other flags are unaffected.)

Sometimes it is useful to know the current flag settings. You can retrieve the current flag values using the flags() function, whose prototype is shown here: fmtflags flags();

This function returns the current value of the flags relative to the invoking stream. The following form of flags() sets the flag values to those specified by flags and returns the previous flag values: fmtflags flags(fmtflags flags); The following program demonstrates flags() and unsetf():

12	

```

// Demonstrate flags() and unsetf().

#include <iostream>
using namespace std;

int main()
{
    ios::fmtflags f;

    f = cout.flags(); ← Get the format flags.

    if(f & ios::showpos)
        cout << "showpos is set for cout.\n";
    else
        cout << "showpos is cleared for cout.\n";

    cout << "\nSetting showpos for cout.\n";
    cout.setf(ios::showpos); ←
    f = cout.flags();        Set the showpos flag.
    if(f & ios::showpos)
        cout << "showpos is set for cout.\n";
    else
        cout << "showpos is cleared for cout.\n";

    cout << "\nClearing showpos for cout.\n";
    cout.unsetf(ios::showpos); ←
    f = cout.flags();        Clear the showpos flag.

    if(f & ios::showpos)
        cout << "showpos is set for cout.\n";
    else
        cout << "showpos is cleared for cout.\n";

    return 0;
}

```


The program produces this output:

```
showpos is cleared for cout.  
Setting showpos for cout.  
showpos is set for cout.  
Clearing showpos for cout.  
showpos is cleared for cout.
```

In the program, notice that the type `fmtflags` is preceded by `ios::` when `f` is declared. This is necessary since `fmtflags` is a type defined by `ios`. In general, whenever you use the name of a type or enumerated constant that is defined by a class, you must qualify it with the name of the class.

Setting the Field Width, Precision, and Fill Character

13	

In addition to the formatting flags, there are three member functions defined by ios that set these additional format values: the field width, the precision, and the fill character. The functions that set these values are `width()`, `precision()`, and `fill()`, respectively. Each is examined in turn.

By default, when a value is output, it occupies only as much space as the number of characters it takes to display it. However, you can specify a minimum field width by using the `width()` function. Its prototype is shown here:

```
streamsize width(streamsize w);
```

Here, `w` becomes the field width, and the previous field width is returned. In some implementations, the field width must be set before each output. If it isn't, the default field width is used. The `streamsize` type is defined as some form of integer by the compiler.

After you set a minimum field width, when a value uses less than the specified width, the field will be padded with the current fill character (space, by default) to reach the field width. If the size of the value exceeds the minimum field width, then the field will be overrun. No values are truncated.

When outputting floating-point values in scientific notation, you can determine the number of digits to be displayed after the decimal point by using the `precision()` function. Its prototype is shown here:

```
streamsize precision(streamsize p);
```

Here, the precision is set to `p`, and the old value is returned. The default precision is 6. In some implementations, the precision must be set before each floating-point output. If you don't set it, the default precision is used.

By default, when a field needs to be filled, it is filled with spaces. You can specify the fill character by using the `fill()` function. Its prototype is

```
char fill(char ch);
```

After a call to `fill()`, `ch` becomes the new fill character, and the old one is returned.

Here is a program that demonstrates these three functions:

14	

```

// Demonstrate width(), precision(), and fill().

#include <iostream>
using namespace std;

int main()
{
    cout.setf(ios::showpos);
    cout.setf(ios::scientific);
    cout << 123 << " " << 123.23 << "\n";

    cout.precision(2); // two digits after decimal point
    cout.width(10);     // in a field of 10 characters
    cout << 123 << " ";
    cout.width(10);     // set width to 10
    cout << 123.23 << "\n";

    cout.fill('#'); // fill using #
    cout.width(10); // in a field of 10 characters
    cout << 123 << " ";
    cout.width(10); // set width to 10
    cout << 123.23;

    return 0;
}

```

Set the precision.

Set field width.

Set the fill character.

The program displays this output:

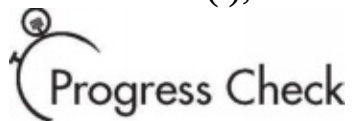
```

+123 +1.232300e+002
      +123 +1.23e+002
#####+123 +1.23e+002

```

As mentioned, in some implementations, it is necessary to reset the field width before each output operation. This is why `width()` is called repeatedly in the preceding program. There are overloaded forms of `width()`, `precision()`, and `fill()` that obtain, but do not change, the current setting. These forms are shown here:

```
char fill( ); streamsize width( ); streamsize precision( );
```



What does `boolalpha` do?

What does `setf()` do?

What function is used to set the fill character?

15	

CRITICAL SKILL 10.5: Using I/O Manipulators

The C++ I/O system includes a second way in which you can alter the format parameters of a stream. This method uses special functions, called manipulators, that can be included in an I/O expression. The standard manipulators are shown in Table 11-1. To use those manipulators that take arguments, you must include `<iomanip>` in your program.

Manipulator	Purpose	Input/Output
<code>boolalpha</code>	Turns on boolalpha flag	Input/Output
<code>dec</code>	Turns on dec flag	Input/Output
<code>endl</code>	Outputs a newline character and flushes the stream	Output
<code>ends</code>	Outputs a null	Output
<code>fixed</code>	Turns on fixed flag	Output
<code>flush</code>	Flushes a stream	Output
<code>hex</code>	Turns on hex flag	Input/Output
<code>internal</code>	Turns on internal flag	Output
<code>left</code>	Turns on left flag	Output
<code>noboolalpha</code>	Turns off boolalpha flag	Input/Output
<code>noshowbase</code>	Turns off showbase flag	Output
<code>noshowpoint</code>	Turns off showpoint flag	Output
<code>noshowpos</code>	Turns off showpos flag	Output
<code>noskipws</code>	Turns off skipws flag	Input
<code>nounitbuf</code>	Turns off unitbuf flag	Output
<code>nouppercase</code>	Turns off uppercase flag	Output

oct	Turns on oct flag	Input/Output
resetiosflags (fmtflags f)	Turns off the flags specified in f	Input/Output
right	Turns on right flag	Output
scientific	Turns on scientific flag	Output
setbase(int base)	Sets the number base to base	Input/Output
setfill(int ch)	Sets the fill character to ch	Output
setiosflags(fmtflags f)	Turns on the flags specified in f	Input/Output
setprecision (int p)	Sets the number of digits of precision	Output
setw(int w)	Sets the field width to w	Output
showbase	Turns on showbase flag	Output
showpoint	Turns on showpoint flag	Output

Table 11-1 The C++ I/O Manipulators

16	

Manipulator	Purpose	Input/Output	A
showpos	Turns on showpos flag	Output	
skipws	Turns on skipws flag	Input	
unitbuf	Turns on unitbuf flag	Output	
uppercase	Turns on uppercase flag	Output	
ws	Skips leading whitespace	Input	

Table 11-1 The C++ I/O Manipulators (continued)

manipulator is used as part of a larger I/O expression. Here is a sample program that uses manipulators to control the format of its output:

// Demonstrate an I/O manipulator.

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    cout << setprecision(2) << 1000.243 << endl;
    cout << setw(20) << "Hello there.";
    return 0;
}
```

Use I/O manipulators.

It produces this output:

```
1e+003
      Hello there.
```


Notice how the manipulators occur in the chain of I/O operations. Also, notice that when a manipulator does not take an argument, such as `endl` in the example, it is not followed by parentheses.

The following program uses `setiosflags()` to set the scientific and `showpos` flags:

```
// Use setiosflags().  
  
#include <iostream>  
#include <iomanip>  
using namespace std;
```

```

int main()
{
    cout << setiosflags(ios::showpos) <<
        setiosflags(ios::scientific) <<
        123 << " " << 123.23;

    return 0;
}

```

← Use `setiosflags()`.

The program shown next uses `ws` to skip any leading whitespace when inputting a string into `s`:

```

// Skip leading whitespace.

#include <iostream>
using namespace std;

int main()
{
    char s[80];

    cin >> ws >> s;
    cout << s;

    return 0;
}

```

← Use `ws`.

CRITICAL SKILL 11.6: Creating Your Own Manipulator Functions

You can create your own manipulator functions. There are two types of manipulator functions: those that take arguments and those that don't. The creation of parameterized manipulators requires the use of techniques beyond the scope of this book. However, the creation of parameterless manipulators is quite easy and is described here.

All parameterless manipulator output functions have this skeleton:

```
ostream &manip_name(ostream &stream)
{
    // your code here

    return stream;
}
```

Here, `manip_name` is the name of the manipulator. It is important to understand that even though the manipulator has as its single argument a pointer to the stream upon which

it is operating, no argument is specified when the manipulator is used in an output expression.

The following program creates a manipulator called `setup()` that turns on left justification, sets the field width to 10, and specifies that the dollar sign will be the fill character.

18	

```
// Create an output manipulator.

#include <iostream>
#include <iomanip>
using namespace std;

ostream &setup(ostream &stream) ← A custom output manipulator
{
    stream.setf(ios::left);
    stream << setw(10) << setfill(' ');
    return stream;
}

int main()
{
    cout << 10 << " " << setup << 10;

    return 0;
}
```

Custom manipulators are useful for two reasons. First, you might need to perform an I/O operation on a device for which none of the predefined manipulators applies—a plotter, for example. In this case, creating your own manipulators will make it more convenient when outputting to the device. Second, you may find that you are repeating the same sequence of operations many times. You can consolidate these operations into a single

manipulator, as the foregoing program illustrates.

All parameterless input manipulator functions have this skeleton:

```
istream &manip_name(istream &stream)
{
    // your code here

    return stream;
}
```

For example, the following program creates the `prompt()` manipulator. It displays a prompting message and then configures input to accept hexadecimal.

19	

```
// Create an input manipulator.

#include <iostream>
#include <iomanip>
using namespace std;

istream &prompt(istream &stream) ← A custom input manipulator
{
    cin >> hex;
    cout << "Enter number using hex format: ";

    return stream;
}

int main()
{
    int i;

    cin >> prompt >> i;
    cout << i;

    return 0;
}
```

Remember that it is crucial that your manipulator return stream. If this is not done, then your manipulator cannot be used in a chain of input or output operations.



What does endl do?

What does ws do?

Is an I/O manipulator used as part of a larger I/O expression?

File I/O

You can use the C++ I/O system to perform file I/O. To perform file I/O, you must include the header `<fstream>` in your program. It defines several important classes and values.

CRITICAL SKILL 11.7: Opening and Closing a File

In C++, a file is opened by linking it to a stream. As you know, there are three types of streams: input, output, and input/output. To open an input stream, you must declare the stream to be of class `ifstream`.

20	

To open an output stream, it must be declared as class `ofstream`. A stream that will be performing both input and output operations must be declared as class `fstream`. For example, this fragment creates one input stream, one output stream, and one stream capable of both input and output:

```
ifstream in; // input
ofstream out; // output
fstream both; // input and output
```

Once you have created a stream, one way to associate it with a file is by using `open()`. This function is a member of each of the three stream classes. The prototype for each is shown here:

```
void ifstream::open(const char *filename,
                    ios::openmode mode = ios::in);
void ofstream::open(const char *filename,
                    ios::openmode mode = ios::out | ios::trunc);
void fstream::open(const char *filename,
                   ios::openmode mode = ios::in | ios::out);
```

Here, `filename` is the name of the file; it can include a path specifier. The value of `mode` determines how the file is opened. It must be one or more of the values defined by `openmode`, which is an enumeration defined by `ios` (through its base class `ios_base`). The values are shown here:

```
ios::app
ios::ate
ios::binary
ios::in
ios::out
ios::trunc
```


You can combine two or more of these values by ORing them together. Including `ios::app` causes all output to that file to be appended to the end. This value can be used only with files capable of output. Including `ios::ate` causes a seek to the end of the file to occur when the file is opened. Although `ios::ate` causes an initial seek to end-of-file, I/O operations can still occur anywhere within the file.

The `ios::in` value specifies that the file is capable of input. The `ios::out` value specifies that the file is capable of output.

The `ios::binary` value causes a file to be opened in binary mode. By default, all files are opened in text mode. In text mode, various character translations may take place, such as carriage return–linefeed sequences being converted into newlines. However, when a file is opened in binary mode, no such character translations will occur. Understand that any file, whether it contains formatted text or raw data, can be opened in either binary or text mode. The only difference is whether character translations take place.

The `ios::trunc` value causes the contents of a preexisting file by the same name to be destroyed, and the file to be truncated to zero length. When creating an output stream using `ofstream`, any preexisting file by that name is automatically truncated.

The following fragment opens a text file for output:

```
ofstream mystream; mystream.open("test");
```

Since the mode parameter to `open()` defaults to a value appropriate to the type of stream being opened, there is often no need to specify its value in the preceding example. (Some compilers do not default the mode parameter for `fstream::open()` to `in | out`, so you might need to specify this explicitly.)

If `open()` fails, the stream will evaluate to false when used in a Boolean expression. You can make use of this fact to confirm that the open operation succeeded by using a statement like this:

```
if(!mystream) {  
  
    cout << "Cannot open file.\n";  
  
    // handle error }
```

In general, you should always check the result of a call to `open()` before attempting to access the file. You can also check to see if you have successfully opened a file by using the `is_open()` function, which is a member of `fstream`, `ifstream`, and `ofstream`. It has this prototype:

```
bool is_open( );
```

It returns true if the stream is linked to an open file and false otherwise. For example, the following checks if `mystream` is currently open:

```
if(!mystream.is_open()) {  
  
    cout << "File is not open.\n";  
  
    // ...
```

Although it is entirely proper to use the `open()` function for opening a file, most of the time you will not do so because the `ifstream`, `ofstream`, and `fstream` classes have constructors that automatically open the file. The constructors have the same parameters and defaults as the `open()` function.

Therefore, the most common way you will see a file opened is shown in this example:

```
ifstream mystream("myfile"); // open file for input
```

If, for some reason, the file cannot be opened, the value of the associated stream variable will evaluate to false. To close a file, use the member function `close()`. For example, to close the file linked to a stream called `mystream`, you would use this statement:

```
mystream.close();
```

The `close()` function takes no parameters and returns no value.


22	


CRITICAL SKILL 11.8: Reading and Writing Text Files


The easiest way to read from or write to a text file is to use the << and >> operators. For example, this program writes an integer, a floating-point value, and a string to a file called test:

```
// Write to file.

#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    ofstream out("test");  Create and open a file called
                           "test" for text output.
    if(!out) {
        cout << "Cannot open file.\n";
        return 1;
    }

    out << 10 << " " << 123.23 << "\n";  Output to the file.
    out << "This is a short text file.";

    out.close();  Close the file.
    return 0;
}
```

The following program reads an integer, a float, a character, and a string from the file created by the previous program:

23	

```

// Read from file.

#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    char ch;
    int i;
    float f;
    char str[80];

    ifstream in("test"); ← Open a file for text input.
    if(!in) {
        cout << "Cannot open file.\n";
        return 1;
    }

    in >> i;
    in >> f;
    in >> ch;
    in >> str; ← Read from the file.

    cout << i << " " << f << " " << ch << "\n";
    cout << str;
    in.close(); ← Close the file.
    return 0;
}

```

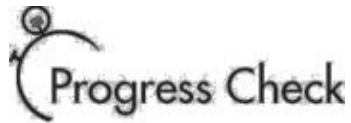
Keep in mind that when the >> operator is used for reading text files, certain character translations occur. For example, whitespace characters are omitted. If you want to prevent any character translations, you must open a file for binary access. Also remember that when >> is used to read a string, input stops when the first whitespace character is encountered.

Ask the Expert

Q: As you explained in Module 1, C++ is a superset of C. I know that C defines an I/O system of its own. Is the C I/O system available to C++ programmers? If so, should it be used in C++ programs?

A: The answer to the first question is yes. The C I/O system is available to C++ programmers. The

answer to the second question is a qualified no. The C I/O system is not object-oriented. Thus, you will nearly always find the C++ I/O system more compatible with C++ programs. However, the C I/O system is still widely used and is quite streamlined, carrying little overhead. Thus, for some highly specialized programs, the C I/O system might be a good choice. Information on the C I/O system can be found in my book C++: The Complete Reference (Osborne/McGraw-Hill).



What class creates an input file?

What function opens a file?

Can you read and write to a file using `<<` and `>>`?

CRITICAL SKILL 11.9: Unformatted and Binary I/O

While reading and writing formatted text files is very easy, it is not always the most efficient way to handle files. Also, there will be times when you need to store unformatted (raw) binary data, not text. The functions that allow you to do this are described here.

When performing binary operations on a file, be sure to open it using the `ios::binary` mode specifier. Although the unformatted file functions will work on files opened for text mode, some character translations may occur. Character translations negate the purpose of binary file operations.

In general, there are two ways to write and read unformatted binary data to or from a file. First, you can write a byte using the member function `put()`, and read a byte using the member function `get()`. The second way uses the block I/O functions: `read()` and `write()`. Each is examined here.

Using `get()` and `put()`

The `get()` function has many forms, but the most commonly used version is shown next, along with that of `put()`:

```
istream &get(char &ch); ostream &put(char ch);
```

The `get()` function reads a single character from the associated stream and

puts that value in ch. It returns a reference to the stream. This value will be null if the end of the file is reached. The put() function writes ch to the stream and returns a reference to the stream.

The following program will display the contents of any file on the screen. It uses the get() function:

25	

```
// Display a file using get().

#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
    char ch;

    if(argc!=2) {
        cout << "Usage: PR <filename>\n";
        return 1;
    }

    ifstream in(argv[1], ios::in | ios::binary);
    if(!in) {
        cout << "Cannot open file.\n";
        return 1;
    }

    while(in) { // in will be false when eof is reached
        in.get(ch);
        if(in) cout << ch;
    }
    in.close();

    return 0;
}
```

Open the file for binary operations.

Read data until the end of the file is reached.

Look closely at the while loop. When in reaches the end of the file, it will be false, causing the while loop to stop.

There is actually a more compact way to code the loop that reads and displays a file, as shown here:

```
while(in.get(ch)) cout << ch;
```

This form works because get() returns the stream in, and in will be false when the end of the file is encountered. This program uses put() to write a string to a file.

```

// Use put() to write to a file.

#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    char *p = "hello there\n";

    ofstream out("test", ios::out | ios::binary);
    if(!out) {
        cout << "Cannot open file.\n";
        return 1;
    }

    // Write characters until the null-terminator is reached.
    while(*p) out.put(*p++); ←
    out.close();           Write a string to a file using put().
                           No character translations will occur.
    return 0;
}

```

After this program executes, the file test will contain the string “hello there” followed by a newline character. No character translations will have taken place.

Reading and Writing Blocks of Data

To read and write blocks of binary data, use the `read()` and `write()` member functions. Their prototypes are shown here:

```
istream &read(char *buf, streamsize num); ostream &write(const char *buf,  
streamsize num);
```

The `read()` function reads `num` bytes from the associated stream and puts them in the buffer pointed to by `buf`. The `write()` function writes `num` bytes to the associated stream from the buffer pointed to by `buf`. As mentioned earlier, `streamsize` is some form of integer defined by the C++ library. It is capable of holding the largest number of bytes that can be transferred in any one I/O operation.

The following program writes and then reads an array of integers:

27	

```

// Use read() and write().

#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    int n[5] = {1, 2, 3, 4, 5};
    register int i;

    ofstream out("test", ios::out | ios::binary);
    if(!out) {
        cout << "Cannot open file.\n";
        return 1;
    }

    out.write((char *) &n, sizeof n); ← Write a block of data.

    out.close();

    for(i=0; i<5; i++) // clear array
        n[i] = 0;

    ifstream in("test", ios::in | ios::binary);
    if(!in) {
        cout << "Cannot open file.\n";
        return 1;
    }
    in.read((char *) &n, sizeof n); ← Read a block of data.

    for(i=0; i<5; i++) // show values read from file
        cout << n[i] << " ";

    in.close();

    return 0;
}

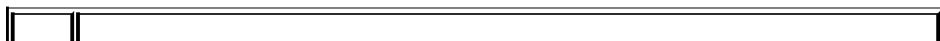
```

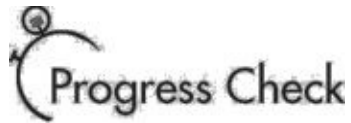
Note that the type casts inside the calls to `read()` and `write()` are necessary when operating on a buffer that is not defined as a character array.

If the end of the file is reached before `num` characters have been read, then `read()` simply stops, and the buffer will contain as many characters as were available. You can find out how many characters have been read using another member function, called `gcount()`, which has this prototype:

```
streamsize gcount( );
```

`gcount()` returns the number of characters read by the last input operation.





To read or write binary data, you open a file using what mode specifier?

What does `get()` do? What does `put()` do?

What function reads a block of data?

CRITICAL SKILL 11.10: More I/O Functions

The C++ I/O system defines other I/O related functions, several of which you will find useful. They are discussed here.

More Versions of `get()`

In addition to the form shown earlier, the `get()` function is overloaded in several different ways. The prototypes for the three most commonly used overloaded forms are shown here:

```
istream &get(char *buf, streamsize num); istream &get(char *buf,  
streamsize num, char delim); int get( );
```

The first form reads characters into the array pointed to by `buf` until either `num-1` characters have been read, a newline is found, or the end of the file has been encountered. The array pointed to by `buf` will be null-terminated by `get()`. If the newline character is encountered in the input stream, it is not extracted. Instead, it remains in the stream until the next input operation.

The second form reads characters into the array pointed to by `buf` until either `num-1` characters have been read, the character specified by `delim` has been found, or the end of the file has been encountered. The array pointed to by `buf` will be null-terminated by `get()`. If the delimiter character

is encountered in the input stream, it is not extracted. Instead, it remains in the stream until the next input operation.

The third overloaded form of `get()` returns the next character from the stream. It returns EOF (a value that indicates end-of-file) if the end of the file is encountered. EOF is defined by `<iostream>`.

One good use for `get()` is to read a string that contains spaces. As you know, when you use `>>` to read a string, it stops reading when the first whitespace character is encountered. This makes `>>` useless for reading a string containing spaces. However, you can overcome this problem by using `get(buf, num)`, as illustrated in this program:

29	

```

// Use get() to read a string that contains spaces.

#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    char str[80];

    cout << "Enter your name: ";
    cin.get(str, 79); ← Use get() to read a string
                        that contains whitespace.
    cout << str << "\n";

    return 0;
}

```

Here, the delimiter to `get()` is allowed to default to a newline. This makes `get()` act much like the standard `gets()` function.

getline()

Another function that performs input is `getline()`. It is a member of each input stream class. Its prototypes are shown here:

```
istream &getline(char *buf, streamsize num); istream &getline(char *buf,
```

streamsize num, char delim);

The first form reads characters into the array pointed to by buf until either num-1 characters have been read, a newline character has been found, or the end of the file has been encountered.

The array pointed to by buf will be null-terminated by getline(). If the newline character is encountered in the input stream, it is extracted, but is not put into buf.

The second form reads characters into the array pointed to by buf until either num-1 characters have been read, the character specified by delim has been found, or the end of the file has been encountered. The array pointed to by buf will be null-terminated by getline(). If the delimiter character is encountered in the input stream, it is extracted, but is not put into buf.

As you can see, the two versions of getline() are virtually identical to the get(buf, num) and get(buf, num, delim) versions of get(). Both read characters from input and put them into the array pointed to by buf until either num-1 characters have been read or until the delimiter character is encountered. The difference between get() and getline() is that getline() reads and removes the delimiter from the input stream; get() does not.

Detecting EOF

30	

You can detect when the end of the file is reached by using the member function `eof()`, which has this prototype:

```
bool eof( );
```

It returns true when the end of the file has been reached; otherwise it returns false.

`peek()` and `putback()`

You can obtain the next character in the input stream without removing it from that stream by using `peek()`. It has this prototype:

```
int peek( );
```

`peek()` returns the next character in the stream, or EOF if the end of the file is encountered. The character is contained in the low-order byte of the return value. You can return the last character read from a stream to that stream by using `putback()`. Its prototype is shown here:

```
istream &putback(char c);
```

where `c` is the last character read.

`flush()`

When output is performed, data is not immediately written to the physical device linked to the stream. Instead, information is stored in an internal buffer until the buffer is full. Only then are the contents of that buffer written to disk. However, you can force the information to be physically written to disk before the buffer is full by calling `flush()`. Its prototype is shown here:

```
ostream &flush( );
```

Calls to `flush()` might be warranted when a program is going to be used in adverse environments (in situations where power outages occur frequently, for example).

NOTE: Closing a file or terminating a program also flushes all buffers.

Project 11-1 A File Comparison Utility

This project develops a simple, yet useful file comparison utility. It works

by opening both files to be compared and then reading and comparing each corresponding set of bytes. If a mismatch is found, the files differ. If the end of each file is reached at the same time and if no mismatches have been found, then the files are the same.

Step by Step

31	

Create a file called CompFiles.cpp.

Begin by adding these lines to CompFiles.cpp:

```
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
    register int i;

    unsigned char buf1[1024], buf2[1024];

    if(argc!=3) {
        cout << "Usage: compfiles <file1> <file2>\n";
        return 1;
    }
```

Notice that the names of the files to compare are specified on the command line.

Add the code that opens the files for binary input operations, as shown here:

```

ifstream f1(argv[1], ios::in | ios::binary);
if(!f1) {
    cout << "Cannot open first file.\n";
    return 1;
}

ifstream f2(argv[2], ios::in | ios::binary);
if(!f2) {
    cout << "Cannot open second file.\n";
    return 1;
}

```

The files are opened for binary operations to prevent the character translations that might occur in text mode.

Add the code that actually compares the files, as shown next:

```

cout << "Comparing files...\n";

do {
    f1.read((char *) buf1, sizeof buf1);
    f2.read((char *) buf2, sizeof buf2);

    if(f1.gcount() != f2.gcount()) {
        cout << "Files are of differing sizes.\n";
        f1.close();
    }
} while(f1.gcount() > 0);

```


32	

```

        f2.close();
        return 0;
    }

    // compare contents of buffers
    for(i=0; i<f1.gcount(); i++)
        if(buf1[i] != buf2[i]) {
            cout << "Files differ.\n";
            f1.close();
            f2.close();
            return 0;
        }

} while(!f1.eof() && !f2.eof());

cout << "Files are the same.\n";

```

This code reads one buffer at a time from each of the files using the `read()` function. It then compares the contents of the buffers. If the contents differ, the files are closed, the “Files differ.” message is displayed, and the program terminates. Otherwise, buffers continue to be read and compared until the end of one (or both) files is reached. Because less than a full buffer may be read at the end of a file, the program uses the `gcount()` function to determine precisely how many characters are in the buffers. If one of the files is shorter than the

other, the values returned by `gcount()` will differ when the end of one of the files is reached. In this case, the message “Files are of differing sizes.” will be displayed. Finally, if the files are the same, then when the end of one file is reached, the other will also have been reached. This is confirmed by calling `eof()` on each stream. If the files compare equal in all regards, then they are reported as equal.

Finish the program by closing the files, as shown here: `f1.close();`

`f2.close(); return 0; }`

The entire `FileComp.cpp` program is shown here:

```
/*
    Project 11-1

    Create a file comparison utility.
*/

#include <iostream>

#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
    register int i;

    unsigned char buf1[1024], buf2[1024];
```

33	

```

if(argc!=3) {
    cout << "Usage: compfiles <file1> <file2>\n";
    return 1;
}

ifstream f1(argv[1], ios::in | ios::binary);
if(!f1) {
    cout << "Cannot open first file.\n";
    return 1;
}
ifstream f2(argv[2], ios::in | ios::binary);
if(!f2) {
    cout << "Cannot open second file.\n";
    return 1;
}

cout << "Comparing files...\n";

do {
    f1.read((char *) buf1, sizeof buf1);
    f2.read((char *) buf2, sizeof buf2);

    if(f1.gcount() != f2.gcount()) {
        cout << "Files are of differing sizes.\n";
        f1.close();
        f2.close();
        return 0;
    }

    // compare contents of buffers
    for(i=0; i<f1.gcount(); i++)
        if(buf1[i] != buf2[i]) {
            cout << "Files differ.\n";
            f1.close();
            f2.close();
            return 0;
        }

} while(!f1.eof() && !f2.eof());

cout << "Files are the same.\n";

f1.close();
f2.close();

return 0;
}

```


34	

To try CompFiles, first copy CompFiles.cpp to a file called temp.txt. Then, try this command line:

```
CompFiles CompFiles.temp txt
```

The program will report that the files are the same. Next, compare CompFiles.cpp to a different file, such as one of the other program files from this module. You will see that CompFiles reports that the files differ.

On your own, try enhancing CompFiles with various options. For example, add an option that ignores the case of letters. Another idea is to have CompFiles display the position within the file where the files differ.

CRITICAL SKILL 11.11: Random Access

So far, files have been read or written sequentially, but you can also access a file in random order. In C++'s I/O system, you perform random access using the `seekg()` and `seekp()` functions. Their most common forms are shown here:

```
istream &seekg(off_type offset, seekdir origin);  
ostream &seekp(off_type offset, seekdir origin);
```

Here,

Value	Meaning
<code>ios::beg</code>	Beginning of file
<code>ios::cur</code>	Current location
<code>ios::end</code>	End of file

`off_type` is an integer type defined by `ios` that is capable of containing the largest valid value that offset can have. `seekdir` is an enumeration that has these values:

The C++ I/O system manages two pointers associated with a file. One is the get pointer, which specifies where in the file the next input operation will occur. The other is the put pointer, which specifies where in the file the next output operation will occur. Each time an input or an output operation takes place, the appropriate pointer is automatically advanced. Using the `seekg()` and `seekp()` functions, it is possible to move this pointer and access the file in a non-sequential fashion.

The `seekg()` function moves the associated file's current get pointer offset number of bytes from the specified origin. The `seekp()` function moves the associated file's current put pointer offset number of bytes from the specified origin.

Generally, random access I/O should be performed only on those files opened for binary operations. The character translations that may occur on text files could cause a position request to be out of sync with the actual contents of the file.

The following program demonstrates the `seekp()` function. It allows you to specify a filename on the command line, followed by the specific byte that you want to change in the file. The program then writes an X at the specified location. Notice that the file must be opened for read/write operations.

```
// Demonstrate random access.

#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;

int main(int argc, char *argv[])
{
    if(argc!=3) {
        cout << "Usage: CHANGE <filename> <byte>\n";
        return 1;
    }

    fstream out(argv[1], ios::in | ios::out | ios::binary);
    if(!out) {
        cout << "Cannot open file.\n";
        return 1;
    }

    out.seekp(atoi(argv[2]), ios::beg);
    out.put('X');
    out.close();

    return 0;
}
```

Seek to a specific byte within the file.
This moves the put pointer.

The next program uses `seekg()`. It displays the contents of a file, beginning with the location you specify on the command line.

36	

```

// Display a file from a given starting point.

#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;

int main(int argc, char *argv[])
{
    char ch;

    if(argc!=3) {
        cout << "Usage: NAME <filename> <starting location>\n";
        return 1;
    }

    ifstream in(argv[1], ios::in | ios::binary);
    if(!in) {
        cout << "Cannot open file.\n";
        return 1;
    }

    in.seekg(atoi(argv[2]), ios::beg); ← This moves the get pointer.

    while(in.get(ch))
        cout << ch;

    return 0;
}

```

You can determine the current position of each file pointer using these functions:

```
pos_type tellg( ); pos_type tellp( );
```

Here, `pos_type` is a type defined by `ios` that is capable of holding the largest value that either function can return. There are overloaded versions of `seekg()` and `seekp()` that move the file pointers to the location specified by the return values of `tellg()` and `tellp()`. Their prototypes are shown here:

```
istream &seekg(pos_type position); ostream &seekp(pos_type position);
```



What function detects the end of the file?

What does `getline()` do?

What functions handle random access position requests?

CRITICAL SKILL 11.12: Checking I/O Status

37	

The C++ I/O system maintains status information about the outcome of each I/O operation. The current status of an I/O stream is described in an object of type `iostate`, which is an enumeration defined by `ios` that includes these members.

Name	Meaning
<code>ios::goodbit</code>	No error bits set
<code>ios::eofbit</code>	1 when end-of-file is encountered; 0 otherwise
<code>ios::failbit</code>	1 when a (possibly) nonfatal I/O error has occurred; 0 otherwise
<code>ios::badbit</code>	1 when a fatal I/O error has occurred; 0 otherwise

There are two ways in which you can obtain I/O status information. First, you can call the `rdstate()`

function. It has this prototype:

```
iostate rdstate( );
```

It returns the current status of the error flags. As you can probably guess from looking at the preceding list of flags, `rdstate()` returns `goodbit` when no error has occurred. Otherwise, an error flag is turned on.

The other way you can determine if an error has occurred is by using one or more of these `ios` member functions:

```
bool bad( ); bool eof( );
```

```
bool fail( ); bool good( );
```

The `eof()` function was discussed earlier. The `bad()` function returns true if `badbit` is set. The `fail()` function returns true if `failbit` is set. The `good()` function returns true if there are no errors. Otherwise they return false.

Once an error has occurred, it may need to be cleared before your program continues. To do this, use the `ios` member function `clear()`, whose prototype is shown here:

```
void clear(iostate flags = ios::goodbit);
```

If flags is goodbit (as it is by default), all error flags are cleared. Otherwise, set flags to the settings you desire.

Before moving on, you might want to experiment with using these status-reporting functions to add extended error-checking to the preceding file examples.

Module 11 Mastery Check

What are the four predefined streams called?

38	

Does C++ define both 8-bit and wide-character streams?

Show the general form for overloading an inserter.

What does `ios::scientific` do?

What does `width()` do?

An I/O manipulator is used within an I/O expression. True or false?

Show how to open a file for reading text input.

Show how to open a file for writing text output.

What does `ios::binary` do?

When the end of the file is reached, the stream variable will evaluate as false. True or false?

Assuming a file is associated with an input stream called `strm`, show how to read to the end of the file.

Write a program that copies a file. Allow the user to specify the name of the input and output file on the command line. Make sure that your program can copy both text and binary files.

Write a program that merges two text files. Have the user specify the names of the two files on the command line in the order they should appear in the output file. Also, have the user specify the name of the output file. Thus, if the program is called `merge`, then the following command line will merge the files `MyFile1.txt` and `MyFile2.txt` into `Target.txt`:

```
merge MyFile1.txt MyFile2.txt Target.txt
```

Show how the `seekg()` statement will seek to the 300th byte in a stream called `MyStrm`.

39	

Module12

Exceptions, Templates, and Other Advanced Topics

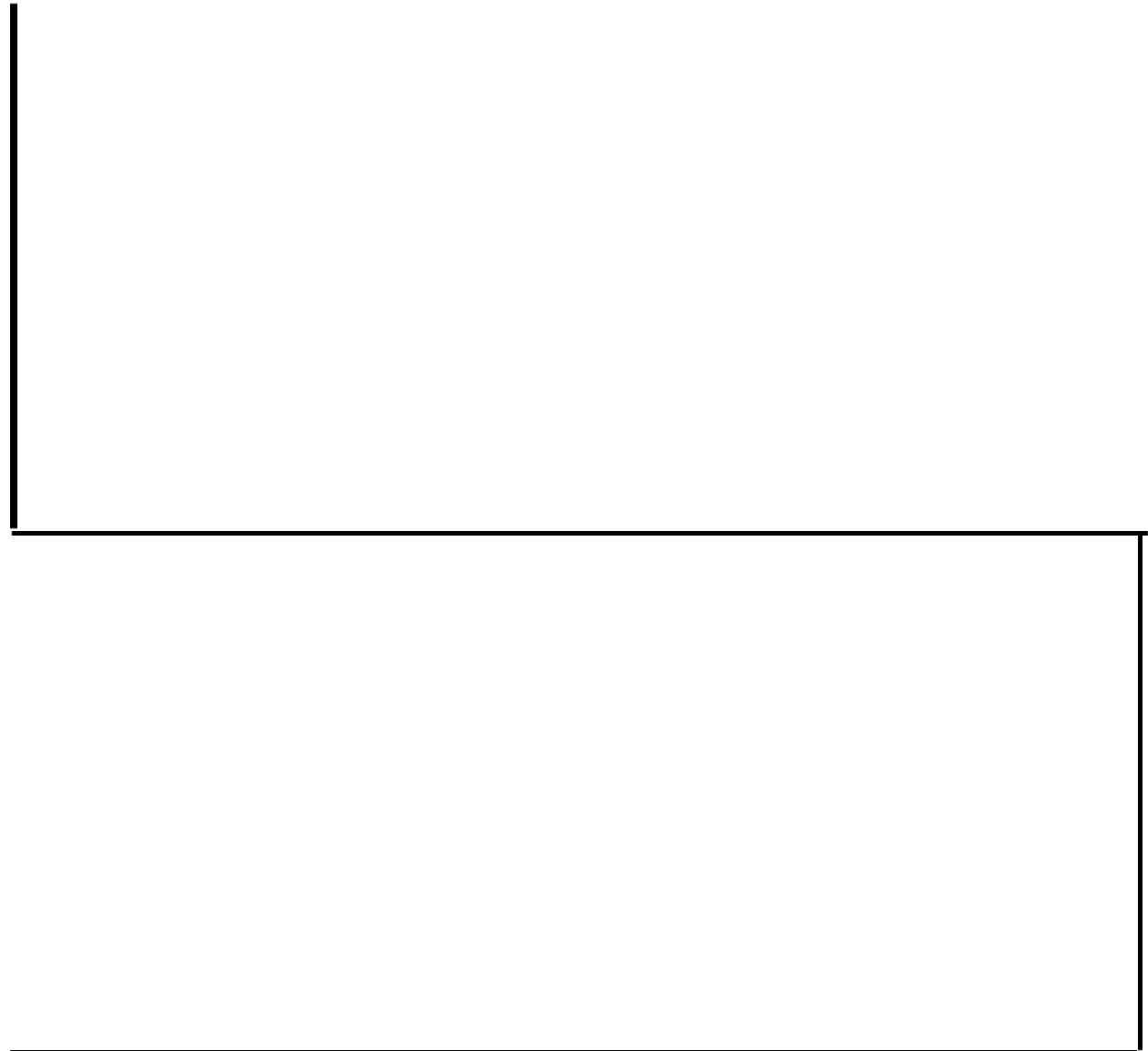


Table of Contents

CRITICAL SKILL 12.1: Exception Handling	2
---	---



CRITICAL SKILL 12.2: Generic Functions	14
CRITICAL SKILL 12.3: Generic Classes	19
CRITICAL SKILL 12.4: Dynamic Allocation	26
CRITICAL SKILL 12.5: Namespaces	35
CRITICAL SKILL 12.6: static Class Members	42
CRITICAL SKILL 12.7: Runtime Type Identification (RTTI)	46
CRITICAL SKILL 12.8: The Casting Operators	49

You have come a long way since the start of this book. In this, the final module, you will examine several important, advanced C++ topics, including exception handling, templates, dynamic allocation, and namespaces. Runtime type ID and the casting operators are also covered. Keep in mind that C++ is a large, sophisticated, professional programming language, and it is not possible to cover every advanced feature, specialized technique, or programming nuance in this beginner's guide. When you finish this module, however, you will have mastered the core elements of the language and will be able to begin writing real-world programs.

1	

CRITICAL SKILL 12.1: Exception Handling

An exception is an error that occurs at runtime. Using C++'s exception handling subsystem, you can, in a structured and controlled manner, handle runtime errors. When exception handling is employed, your program automatically invokes an error-handling routine when an exception occurs. The principal advantage of exception handling is that it automates much of the error-handling code that previously had to be entered “by hand” into any large program.

Exception Handling Fundamentals

C++ exception handling is built upon three keywords: try, catch, and throw. In the most general terms, program statements that you want to monitor for exceptions are contained in a try block. If an exception (that is, an error) occurs within the try block, it is thrown (using throw). The exception is caught, using catch, and processed. The following discussion elaborates upon this general description.

Code that you want to monitor for exceptions must have been executed from within a try block. (A function called from within a try block is also monitored.) Exceptions that can be thrown by the monitored code are caught by a catch statement that immediately follows the try statement in which the exception was thrown. The general forms of try and catch are shown here:

```
try {  
    // try block  
}  
catch (type1 arg) {  
    // catch block  
}  
catch (type2 arg) {  
    // catch block  
}  
catch (type3 arg) {  
    // catch block  
}  
// ...  
catch (typeN arg) {  
    // catch block  
}
```

The try block must contain the portion of your program that you want to monitor for errors. This section can be as short as a few statements within one function, or as all-encompassing as a try block that encloses the main() function code (which would, in effect, cause the entire program to be monitored).

When an exception is thrown, it is caught by its corresponding catch statement, which then processes the exception. There can be more than one catch statement associated with a try. The type of the exception determines which catch statement is used. That is, if the data type specified by a catch statement matches that of the exception, then that catch statement is executed (and all others are bypassed). When an exception is caught, arg will receive its value. Any type of data can be caught, including classes that you create.

2	

The general form of the throw statement is shown here:

`throw exception;`

`throw` generates the exception specified by `exception`. If this exception is to be caught,

Exceptions, Templates, and Other Advanced Topics

then `throw` must be executed either from within a `try` block itself, or from any function called from within the `try` block (directly or indirectly).

If an exception is thrown for which there is no applicable catch statement, an abnormal program termination will occur. That is, your program will stop abruptly in an uncontrolled manner. Thus, you will want to catch all exceptions that will be thrown.

Here is a simple example that shows how C++ exception handling operates:

```
// A simple exception handling example.

#include <iostream>
using namespace std;

int main()
{
    cout << "start\n";

    try { // start a try block ←————— Begin a try block.
        cout << "Inside try block\n";
        throw 99; // throw an error ←————— Throw an exception.
        cout << "This will not execute";
    }
    catch (int i) { // catch an error ←————— Catch the exception.
        cout << "Caught an exception -- value is: ";
        cout << i << "\n";
    }

    cout << "end";

    return 0;
}
```

This program displays the following output:

```
start Inside  
try block  
Caught an exception -- value is: 99  
end
```

Look carefully at this program. As you can see, there is a try block containing three statements and a catch(int i) statement that processes an integer exception. Within the try block, only two of the three statements will execute: the first cout statement and the throw. Once an exception has been thrown, control passes to the catch expression, and the try block is terminated. That is, catch is not called.

3	

Rather, program execution is transferred to it. (The program's stack is automatically reset, as necessary, to accomplish this.) Thus, the cout statement following the throw will never execute.

Usually, the code within a catch statement attempts to remedy an error by taking appropriate action. If the error can be fixed, then execution will continue with the statements following the catch. Otherwise, program execution should be terminated in a controlled manner.

As mentioned earlier, the type of the exception must match the type specified in a catch statement. For example, in the preceding program, if you change the type in the catch statement to double, then the exception will not be caught and abnormal termination will occur. This change is shown here:

```
// This example will not work.

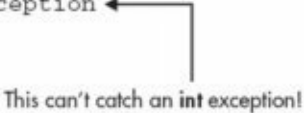
#include <iostream>
using namespace std;

int main()
{
    cout << "start\n";

    try { // start a try block
        cout << "Inside try block\n";
        throw 99; // throw an error
        cout << "This will not execute";
    }
    catch (double i) { // won't work for an int exception ←
        cout << "Caught an exception -- value is: ";
        cout << i << "\n";
    }

    cout << "end";

    return 0;
}
```



This can't catch an **int** exception!

This program produces the following output because the integer exception will not be caught by the `catch(double i)` statement. Of course, the final message indicating abnormal termination will vary from compiler to compiler.

```
start Inside  
try block  
Abnormal program termination
```

An exception thrown by a function called from within a try block can be handled by that try block. For example, this is a valid program:

4	

```

/* Throwing an exception from a function called
   from within a try block. */

#include <iostream>
using namespace std;

void Xtest(int test)
{
    cout << "Inside Xtest, test is: " << test << "\n";
    if(test) throw test;
}

int main()
{
    cout << "start\n";

    try { // start a try block
        cout << "Inside try block\n";
        Xtest(0);
        Xtest(1);
        Xtest(2);
    }
    catch (int i) { // catch an error
        cout << "Caught an exception -- value is: ";
        cout << i << "\n";
    }

    cout << "end";

    return 0;
}

```

← This exception is caught by
the catch statement in main().

← Because Xtest() is called from
within a try block, its code is
also monitored for errors.

This program produces the following output:

```
start
Inside try block
Inside Xtest, test is: 0
Inside Xtest, test is: 1
Caught an exception -- value is: 1
end
```

As the output confirms, the exception thrown in `Xtest()` was caught by the exception handler in `main()`. A try block can be localized to a function. When this is the case, each time the function is entered, the exception handling relative to that function is reset. Examine this sample program:

5	

```

// A try block can be localized to a function.

#include <iostream>
using namespace std;

// A try/catch is reset each time a function is entered.
void Xhandler(int test)
{
    try{ ←———— This try block is local to Xhandler().
        if(test) throw test;
    }
    catch(int i) {
        cout << "Caught One!  Ex. #: " << i << '\n';
    }
}

int main()
{
    cout << "start\n";

    Xhandler(1);
    Xhandler(2);

// A try block can be localized to a function.

#include <iostream>
using namespace std;

// A try/catch is reset each time a function is entered.
void Xhandler(int test)
{
    try{ ←———— This try block is local to Xhandler().
        if(test) throw test;
    }
    catch(int i) {
        cout << "Caught One!  Ex. #: " << i << '\n';
    }
}

int main()
{
    cout << "start\n";

    Xhandler(1);
    Xhandler(2);
    Xhandler(0);
    Xhandler(3);

    cout << "end";

    return 0;
}

```


This program displays the following output:

start

6	

```
Caught One! Ex. #: 1  
Caught One! Ex. #: 2  
Caught One! Ex. #: 3  
end
```

In this example, three exceptions are thrown. After each exception, the function returns. When the function is called again, the exception handling is reset. In general, a try block is reset each time it is entered. Thus, a try block that is part of a loop will be reset each time the loop repeats.



In the language of C++, what is an exception?

Exception handling is based on what three keywords?

An exception is caught based on its type.
True or false?

Using Multiple catch Statements

As stated earlier, you can associate more than one catch statement with a try. In fact, it is common to do so. However, each catch must catch a different type of exception. For example, the program shown next catches both integers and character pointers.


```

// Use multiple catch statements.

#include <iostream>
using namespace std;

// Different types of exceptions can be caught.
void Xhandler(int test)
{
    try(
        if(test) throw test; // throw int
        else throw "Value is zero"; // throw char *
    )
    catch(int i) { ←————— This catches int exceptions.
        cout << "Caught One!  Ex. #: " << i << '\n';
    }
    catch(char *str) { ←————— This catches char * exceptions.
        cout << "Caught a string: ";
        cout << str << '\n';
    }
}

```



```

int main()
{
    cout << "start\n";

    Xhandler(1);
    Xhandler(2);
    Xhandler(0);
    Xhandler(3);

    cout << "end";

    return 0;
}

```

In general, catch expressions are checked in the order in which they occur in a program. Only a matching statement is executed. All other catch blocks are ignored.

Catching Base Class Exceptions

There is one important point about multiple catch statements that relates to derived classes. A catch clause for a base class will also match any class derived from that base. Thus, if you want to catch exceptions of both a base class type and a derived class type, put the derived class first in the catch sequence. If you don't, the base class catch will also catch all derived classes. For example, consider the following program:

8	

```

// Catching derived classes. This program is wrong!

#include <iostream>
using namespace std;

class B {
};

class D: public B {
};

int main()
{
    D derived;

    try {
        throw derived;
    }
    catch(B b) { ←————— This catch list is in the
        cout << "Caught a base class.\n";           wrong order! You must
    }                                                 catch derived classes
    catch(D d) { ←————— before base classes.
        cout << "This won't execute.\n";
    }

    return 0;
}

```

Here, because `derived` is an object that has `B` as a base class, it will be caught by the first catch clause, and the second clause will never execute. Some compilers will flag this condition with a warning message. Others may issue an error message and stop compilation. Either way, to fix this condition, reverse the order of the catch clauses.

Catching All Exceptions

In some circumstances, you will want an exception handler to catch all exceptions instead of just a certain type. To do this, use this form of catch:

```
catch(...) { // process all exceptions }
```

Here, the ellipsis matches any type of data. The following program illustrates `catch(...)`:

9	

```

// This example catches all exceptions.

#include <iostream>
using namespace std;

void Xhandler(int test)
{
    try{
        if(test==0) throw test; // throw int
        if(test==1) throw 'a'; // throw char
        if(test==2) throw 123.23; // throw double
    }
    catch(...) { // catch all exceptions ← Catch all exceptions.
        cout << "Caught One!\n";
    }
}

int main()
{
    cout << "start\n";

    Xhandler(0);
    Xhandler(1);
    Xhandler(2);

    cout << "end";

    return 0;
}

```

This program displays the following output:

```
start
Caught One!
Caught One!
Caught One!
end
```

Xhandler() throws three types of exceptions: int, char, and double. All are caught using the catch(...) statement.

One very good use for catch(...) is as the last catch of a cluster of catches. In this capacity, it provides a useful default or “catch all” statement. Using catch(...) as a default is a good way to catch all exceptions that you don’t want to handle explicitly. Also, by catching all exceptions, you prevent an unhandled exception from causing an abnormal program termination.

Specifying Exceptions Thrown by a Function

You can specify the type of exceptions that a function can throw outside of itself. In fact, you can also prevent a function from throwing any exceptions whatsoever. To accomplish these restrictions, you must add a throw clause to a function definition. The general form of this clause is

10	

ret-type func-name(arg-list) throw(type-list) { // ... }

Here, only those data types contained in the comma-separated type-list can be thrown by the function. Throwing any other type of expression will cause abnormal program termination. If you don't want a function to be able to throw any exceptions, then use an empty list.

NOTE: At the time of this writing, Visual C++ does not actually prevent a function from throwing an exception

type that is not specified in the **throw** clause. This is nonstandard behavior. You can still specify a **throw** clause, but such a clause is informational only.

The following program shows how to specify the types of exceptions that can be thrown from a function:

```
// Restricting function throw types.


#include <iostream>
using namespace std;

// This function can only throw ints, chars, and doubles.

void Xhandler(int test) throw(int, char, double)
{
    if(test==0) throw test;    // throw int
    if(test==1) throw 'a';    // throw char
    if(test==2) throw 123.23; // throw double
}

int main()
{
    cout << "start\n";

    try{
        Xhandler(0); // also, try passing 1 and 2 to Xhandler()
```



Specify the exceptions that can be thrown by Xhandler().

11	

```

    }
    catch(int i) {
        cout << "Caught int\n";
    }
    catch(char c) {
        cout << "Caught char\n";
    }
    catch(double d) {
        cout << "Caught double\n";
    }

    cout << "end";

    return 0;
}

```

In this program, the function `Xhandler()` can only throw integer, character, and double exceptions. If it attempts to throw any other type of exception, then an abnormal program termination will occur. To see an example of this, remove `int` from the list and retry the program. An error will result. (As mentioned, currently Visual C++ does not restrict the exceptions that a function can throw.)

It is important to understand that a function can only be restricted in what types of exceptions it throws back to the `try` block that has called it. That is, a `try` block within a function can throw any type of exception, as long as the exception is caught within that function. The restriction applies only when throwing an exception outside of the function.

Rethrowing an Exception

You can rethrow an exception from within an exception handler by calling `throw` by itself, with no exception. This causes the current exception to be passed on to an outer try/catch sequence. The most likely reason for calling `throw` this way is to allow multiple handlers access to the exception. For example, perhaps one exception handler manages one aspect of an exception, and a second handler copes with another aspect. An exception can only be rethrown from within a catch block (or from any function called from within that block). When you rethrow an exception, it will not be recaptured by the same catch statement. It will propagate to the next catch statement. The following program illustrates rethrowing an exception. It rethrows a `char *` exception.

```
// Example of "rethrowing" an exception.
```

```
#include <iostream>
using namespace std;

void Xhandler()
{
    try {
        throw "hello"; // throw a char *
    }
}
```

```

    catch(char *) { // catch a char *
        cout << "Caught char * inside Xhandler\n";
        throw ; // rethrow char * out of function ← Rethrow an exception.
    }
}

int main()
{
    cout << "start\n";

    try{
        Xhandler();
    }
    catch(char *) {
        cout << "Caught char * inside main\n";
    }

    cout << "end";

    return 0;
}

```

This program displays the following output:

start

```
Caught char * inside Xhandler  
Caught char * inside main  
End
```



Show how to catch all exceptions.

How do you specify the type of exceptions that can be thrown out of a function?

How do you rethrow an exception?

Templates

The template is one of C++'s most sophisticated and high-powered features. Although not part of the original specification for C++, it was added several years ago and is supported by all modern C++ compilers. Templates help you achieve one of the most elusive goals in programming: the creation of reusable code.

13	
----	--

Using templates, it is possible to create generic functions and classes. In a generic function or class, the type of data upon which the function or class operates is specified as a parameter. Thus, you can use one function or class with several different types of data without having to explicitly recode specific versions for each data type. Both generic functions and generic classes are introduced here.

Ask the Expert

Q: It seems that there are two ways for a function to report an error: to throw an exception or to return an error code. In general, when should I use each approach?

A: You are correct, there are two general approaches to reporting errors: throwing exceptions and

returning error codes. Today, language experts favor exceptions rather than error codes. For example, both the Java and C# languages rely heavily on exceptions, using them to report most types of common errors, such as an error opening a file or an arithmetic overflow. Because C++ is derived from C, it uses a blend of error codes and exceptions to report errors. Thus, many error conditions that relate to C++ library functions are reported using error return codes. However, in new code that you write, you should consider using exceptions to report errors. It is the way modern code is being written.

CRITICAL SKILL 12.2: Generic Functions

A generic function defines a general set of operations that will be applied to various types of data. The type of data that the function will operate upon is passed to it as a parameter. Through a generic function, a single general procedure can be applied to a wide range of data. As you probably know, many algorithms are logically the same no matter what type of data is being operated upon. For example, the Quicksort sorting algorithm is the same whether it is applied to an array of integers or an array of floats. It is just

that the type of data being sorted is different. By creating a generic function, you can define the nature of the algorithm, independent of any data. Once you have done this, the compiler will automatically generate the correct code for the type of data that is actually used when you execute the function. In essence, when you create a generic function, you are creating a function that can automatically overload itself.

A generic function is created using the keyword `template`. The normal meaning of the word “template” accurately reflects its use in C++. It is used to create a template (or framework) that describes what a function will do, leaving it to the compiler to fill in the details as needed. The general form of a generic function definition is shown here:

```
template <class Ttype> ret-type func-name(parameter list) { // body of function }
```

Here, `Ttype` is a placeholder name for a data type. This name is then used within the function definition to declare the type of data upon which the function operates. The compiler will automatically replace `Ttype` with an actual data type when it creates a specific version of the function. Although the use of the keyword `class` to specify a generic type in a template declaration is traditional, you may also use the keyword `typename`.

The following example creates a generic function that swaps the values of the two variables with which it is called. Because the process of exchanging two values is independent of the type of the variables, it is a good candidate for being made into a generic function.

```
// Function template example.
```

```
#include <iostream>
using namespace std;
```

```
// This is a function template.
```

```
template <class X> void swapargs(X &a, X &b)
```

← A generic function that exchanges the values of its arguments. Here, X is the generic data type.

```
{
    X temp;
    temp = a;
    a = b;
    b = temp;
}
```

```
int main()
```

```
{
    int i=10, j=20;
    float x=10.1, y=23.3;
    char a='x', b='z';
```

```
    cout << "Original i, j: " << i << ' ' << j << '\n';
    cout << "Original x, y: " << x << ' ' << y << '\n';
    cout << "Original a, b: " << a << ' ' << b << '\n';
```

```
    swapargs(i, j); // swap integers
    swapargs(x, y); // swap floats
    swapargs(a, b); // swap chars
```

— The compiler automatically creates versions of `swapargs()` that use the type of data specified by its arguments.

```
    cout << "Swapped i, j: " << i << ' ' << j << '\n';
    cout << "Swapped x, y: " << x << ' ' << y << '\n';
    cout << "Swapped a, b: " << a << ' ' << b << '\n';
```

```
    return 0;
```

```
}
```

Let's look closely at this program. The line

```
template <class X> void swapargs(X &a, X &b)
```

tells the compiler two things: that a template is being created and that a generic definition is beginning. Here, X is a generic type that is used as a placeholder. After the template portion, the function swapargs() is declared, using X as the data type of the values that will be swapped. In main(), the swapargs() function is called using three different types of data: ints, floats, and chars. Because swapargs() is a generic function, the compiler automatically creates three versions of swapargs(): one that will exchange integer values, one that will exchange floating-point values, and one that will swap


characters. Thus, the same generic `swap()` function can be used to exchange arguments of any type of data.

Here are some important terms related to templates. First, a generic function (that is, a function definition preceded by a template statement) is also called a template function. Both terms are used interchangeably in this book. When the compiler creates a specific version of this function, it is said to have created a specialization. This is also called a generated function. The act of generating a function is referred to as instantiating it. Put differently, a generated function is a specific instance of a template function.

A Function with Two Generic Types

You can define more than one generic data type in the template statement by using a comma-separated list. For example, this program creates a template function that has two generic types:

```
#include <iostream>
using namespace std;

template <class Type1, class Type2>  Two generic types
    void myfunc(Type1 x, Type2 y)
    {
        cout << x << ' ' << y << '\n';
    }

int main()
{
    myfunc(10, "hi");

    myfunc(0.23, 10L);

    return 0;
}
```

In this example, the placeholder types Type1 and Type2 are replaced by the compiler with the data types int and char *, and double and long, respectively, when the compiler generates the specific instances of myfunc() within main().

Explicitly Overloading a Generic Function

Even though a generic function overloads itself as needed, you can explicitly overload one, too. This is formally called explicit specialization. If you overload a generic function, then that overloaded function overrides (or “hides”) the generic function relative to that specific version. For example, consider the following, revised version of the argument-swapping example shown earlier:

16	

```
// Specializing a template function.

#include <iostream>
using namespace std;

template <class X> void swapargs(X &a, X &b)
{
    X temp;
```

17	

```

    temp = a;
    a = b;
    b = temp;
    cout << "Inside template swapargs.\n";
}

// This overrides the generic version of swapargs() for ints.
void swapargs(int &a, int &b) ← Explicit overload of swapargs()
{
    int temp;

    temp = a;
    a = b;
    b = temp;
    cout << "Inside swapargs int specialization.\n";
}

int main()
{
    int i=10, j=20;
    float x=10.1, y=23.3;
    char a='x', b='z';

    cout << "Original i, j: " << i << ' ' << j << '\n';
    cout << "Original x, y: " << x << ' ' << y << '\n';
    cout << "Original a, b: " << a << ' ' << b << '\n';

    swapargs(i, j); // calls explicitly overloaded swapargs()
    swapargs(x, y); // calls generic swapargs()
    swapargs(a, b); // calls generic swapargs() ← This calls the explicit
                                                    overload of swapargs().

    cout << "Swapped i, j: " << i << ' ' << j << '\n';
    cout << "Swapped x, y: " << x << ' ' << y << '\n';
    cout << "Swapped a, b: " << a << ' ' << b << '\n';

    return 0;
}

```

This program displays the following output:

```

Original i, j: 10 20
Original x, y: 10.1 23.3
Original a, b: x z
Inside swapargs int specialization.
Inside template swapargs.
Inside template swapargs.
Swapped i, j: 20 10
Swapped x, y: 23.3 10.1
Swapped a, b: z x

```


18	

As the comments inside the program indicate, when `swapargs(i, j)` is called, it invokes the explicitly overloaded version of `swapargs()` defined in the program. Thus, the compiler does not generate this version of the generic `swapargs()` function, because the generic function is overridden by the explicit overloading.

Relatively recently, an alternative syntax was introduced to denote the explicit specialization of a function. This newer approach uses the `template<>` keyword. For example, using the newer specialization syntax, the overloaded `swapargs()` function from the preceding program looks like this:

```
// Use the newer-style specialization syntax.
template<> void swapargs<int>(int &a, int &b)
{
    int temp;

    temp = a;
    a = b;
    b = temp;
    cout << "Inside swapargs int specialization.\n";
}
```

As you can see, the new-style syntax uses the `template<>` construct to indicate specialization. The type of data for which the specialization is being created is placed inside the angle brackets following the function name. This same syntax is used to specialize any type of generic function. While there is no advantage to using one specialization syntax over the other at this time, the new-style syntax is probably a better approach for the long term.

Explicit specialization of a template allows you to tailor a version of a generic function to accommodate a unique situation—perhaps to take

advantage of some performance boost that applies to only one type of data, for example. However, as a general rule, if you need to have different versions of a function for different data types, you should use overloaded functions rather than templates.

CRITICAL SKILL 12.3: Generic Classes

In addition to using generic functions, you can also define a generic class. When you do this, you create a class that defines all the algorithms used by that class; however, the actual type of data being manipulated will be specified as a parameter when objects of that class are created.

Generic classes are useful when a class uses logic that can be generalized. For example, the same algorithm that maintains a queue of integers will also work for a queue of characters, and the same mechanism that maintains a linked list of mailing addresses will also maintain a linked list of auto-part information. When you create a generic class, it can perform the operation you define, such as maintaining a queue or a linked list, for any type of data. The compiler will automatically generate the correct type of object, based upon the type you specify when the object is created.

The general form of a generic class declaration is shown here:

19	

```
template <class Ttype> class class-name {  
  
    // body of class }
```

Here, Ttype is the placeholder type name, which will be specified when a class is instantiated. If necessary, you can define more than one generic data type using a comma-separated list.

Once you have created a generic class, you create a specific instance of that class using the following general form:

```
class-name <type> ob;
```

Here, type is the type name of the data that the class will be operating upon. Member functions of a generic class are, themselves, automatically generic. You need not use template to explicitly specify them as such.

Here is a simple example of a generic class:

```
// A simple generic class.  
  
#include <iostream>  
using namespace std;  
  
template <class T> class MyClass { ← Declare a generic class.  
    T x, y;                               Here, T is the generic type.  
public:  
    MyClass(T a, T b) {  
        x = a;  
        y = b;  
    }  
    T div() { return x/y; }  
};  
  
int main()                                Create a specific instance  
{                                           of a generic class.  
    // Create a version of MyClass for doubles.  
    MyClass<double> d_ob(10.0, 3.0 ); ←  
    cout << "double division: " << d_ob.div() << "\n";  
  
    // Create a version of MyClass for ints.  
    MyClass<int> i_ob(10, 3);  
    cout << "integer division: " << i_ob.div() << "\n";  
  
    return 0;  
}
```

The output is shown here:

double division: 3.33333

20	

integer division: 3

As the output shows, the double object performed a floating-point division, and the int object performed an integer division.

When a specific instance of `MyClass` is declared, the compiler automatically generates versions of the `div()` function, and `x` and `y` variables necessary for handling the actual data. In this example, two different types of objects are declared. The first, `d_ob`, operates on double data. This means that `x` and `y` are double values, and the outcome of the division—and the return type of `div()`—is double. The second, `i_ob`, operates on type `int`. Thus, `x`, `y`, and the return type of `div()` are `int`. Pay special attention to these declarations:

Exceptions, Templates, and Other Advanced Topics

```
MyClass<double> d_ob(10.0, 3.0); MyClass<int> i_ob(10, 3);
```

Notice how the desired data type is passed inside the angle brackets. By changing the type of data specified when `MyClass` objects are created, you can change the type of data operated upon by `MyClass`.

A template class can have more than one generic data type. Simply declare all the data types required by the class in a comma-separated list within the template specification. For instance, the following example creates a class that uses two generic data types:

```

/* This example uses two generic data types in a
   class definition. */
#include <iostream>
using namespace std;

template <class T1, class T2> class MyClass
{
    T1 i;
    T2 j;
public:
    MyClass(T1 a, T2 b) { i = a; j = b; }
    void show() { cout << i << ' ' << j << '\n'; }
};

int main()
{
    MyClass<int, double> ob1(10, 0.23);
    MyClass<char, char *> ob2('X', "This is a test");

    ob1.show(); // show int, double
    ob2.show(); // show char, char *

    return 0;
}

```

This program produces the following output:

21	

10 0.23

X This is a test

The program declares two types of objects. ob1 uses int and double data. ob2 uses a character and a character pointer. For both cases, the compiler automatically generates the appropriate data and functions to accommodate the way the objects are created.

Explicit Class Specializations

As with template functions, you can create a specialization of a generic class. To do so, use the `template<>` construct as you did when creating explicit function specializations. For example:

```

// Demonstrate class specialization.

#include <iostream>
using namespace std;

template <class T> class MyClass {
    T x;
public:
    MyClass(T a) {
        cout << "Inside generic MyClass\n";
        x = a;
    }
    T getx() { return x; }
};

// Explicit specialization for int.
template <> class MyClass<int> {
    int x;
public:
    MyClass(int a) {
        cout << "Inside MyClass<int> specialization\n";
        x = a * a;
    }
    int getx() { return x; }
};

int main()
{
    MyClass<double> d(10.1);
    cout << "double: " << d.getx() << "\n\n";

    MyClass<int> i(5);
    cout << "int: " << i.getx() << "\n";

    return 0;
}

```

← This is an explicit specialization of **MyClass**.

← This uses the explicit specialization of **MyClass**.

This program displays the following output:

22	

Inside generic MyClass

double: 10.1

Inside MyClass<int> specialization

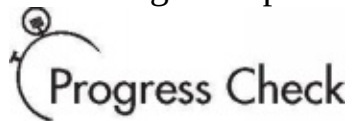
int: 25

In the program, pay close attention to this line:

```
template <> class MyClass<int> {
```

It tells the compiler that an explicit integer specialization of MyClass is being created. This same general syntax is used for any type of class specialization.

Explicit class specialization expands the utility of generic classes because it lets you easily handle one or two special cases while allowing all others to be automatically processed by the compiler. Of course, if you find that you are creating too many specializations, then you are probably better off not using a template class in the first place.



What keyword is used to declare a generic function or class?

Can a generic function be explicitly overloaded?

In a generic class, are all of its member functions also automatically generic?

Project 12-1 Creating a Generic Queue Class

In Project 8-2, you created a Queue class that maintained a queue of characters. In this project, you will convert Queue into a generic class that

can operate on any type of data. Queue is a good choice for conversion to a generic class, because its logic is separate from the data upon which it functions. The same mechanism that stores integers, for example, can also store floating-point values, or even objects of classes that you create. Once you have defined a generic Queue class, you can use it whenever you need a queue.

Step by Step

Begin by copying the Queue class from Project 8-2 into a file called GenericQ.cpp.

Change the Queue declaration into a template, as shown here:

```
template <class QType> class Queue {
```

Here, the generic data type is called QType.

Change the data type of the q array to QType, as shown next:

23	

QType q[maxQsize]; // this array holds the queue

Because q is now generic, it can be used to hold whatever type of data an object of Queue declares.

Change the data type of the parameter to the put() function to QType, as shown here:

```
// Put a data into the queue.
void put(QType data) {
    if(putloc == size) {
        cout << " -- Queue is full.\n";
        return;
    }

    putloc++;
    q[putloc] = data;
}
```

Change the return type of get() to QType, as shown next:

```
// Get data from the queue.
QType get() {
    if(getloc == putloc) {
        cout << " -- Queue is empty.\n";
        return 0;
    }

    getloc++;
    return q[getloc];
}
```

The entire generic Queue class is shown here along with a main() function to demonstrate its use:

```
/*
    Project 12-1

    A template queue class.
*/
#include <iostream>
using namespace std;

const int maxQsize = 100;

// This creates a generic queue class.
template <class QType> class Queue {
    QType q[maxQsize]; // this array holds the queue
    int size; // maximum number of elements that the queue can store
    int putloc, getloc; // the put and get indices
public:

    // Construct a queue of a specific length.
    Queue(int len) {
        // Queue must be less than max and positive.
        if(len > maxQsize) len = maxQsize;
        else if(len <= 0) len = 1;
```

24	


```

    size = len;
    putloc = getloc = 0;
}

// Put data into the queue.
void put(QType data) {
    if(putloc == size) {
        cout << " -- Queue is full.\n";
        return;
    }

    putloc++;
    q[putloc] = data;
}

// Get data from the queue.
QType get() {
    if(getloc == putloc) {
        cout << " -- Queue is empty.\n";
        return 0;
    }

    getloc++;
    return q[getloc];
}

};

// Demonstrate the generic Queue.
int main()
{
    Queue<int> iQa(10), iQb(10); // create two integer queues

    iQa.put(1);
    iQa.put(2);
    iQa.put(3);

    iQb.put(10);
    iQb.put(20);
    iQb.put(30);

    cout << "Contents of integer queue iQa: ";
    for(int i=0; i < 3; i++)
        cout << iQa.get() << " ";

```


25	

```

    cout << endl;

    cout << "Contents of integer queue iQb: ";
    for(int i=0; i < 3; i++)
        cout << iQb.get() << " ";
    cout << endl;

    Queue<double> dQa(10), dQb(10); // create two double queues

    Queue<double> dQa(10), dQb(10); // create two double queues

    dQa.put(1.01);
    dQa.put(2.02);
    dQa.put(3.03);

    dQb.put(10.01);
    dQb.put(20.02);
    dQb.put(30.03);

    cout << "Contents of double queue dQa: ";
    for(int i=0; i < 3; i++)
        cout << dQa.get() << " ";
    cout << endl;

    cout << "Contents of double queue dQb: ";
    for(int i=0; i < 3; i++)
        cout << dQb.get() << " ";
    cout << endl;

    return 0;
}

```

The output is shown here:

```

Contents of integer queue iQa: 1 2 3
Contents of integer queue iQb: 10 20 30
Contents of double queue dQa: 1.01 2.02 3.03
Contents of double queue dQb: 10.01 20.02 30.03

```

As the Queue class illustrates, generic functions and classes are powerful tools that you can use to maximize your programming efforts, because they allow you to define the general form of an object that can then be used with any type of data. You are saved from the tedium of creating separate implementations for each data type for which you want the algorithm to work. The compiler automatically creates the specific versions of the class for you.

CRITICAL SKILL 12.4: Dynamic Allocation

There are two primary ways in which a C++ program can store information in the main memory of the computer. The first is through the use of variables. The storage provided by variables is fixed at compile

26	

time and cannot be altered during the execution of a program. The second way information can be stored is through the use of C++'s dynamic allocation system. In this method, storage for data is allocated as needed from the free memory area that lies between your program (and its permanent storage area) and the stack. This region is called the heap. (Figure 12-1 shows conceptually how a C++ program appears in memory.)

Dynamically allocated storage is determined at runtime. Thus, dynamic allocation makes it possible for your program to create variables that it needs during its execution. It can create as many or as few variables as required, depending upon the situation. Dynamic allocation is often used to support such data structures as linked lists, binary trees, and sparse arrays. Of course, you are free to use dynamic allocation wherever you determine it to be of value. Dynamic allocation for one purpose or another is an important part of nearly all real-world programs.

Memory to satisfy a dynamic allocation request is taken from the heap. As you might guess, it is possible, under fairly extreme cases, for free memory to become exhausted. Therefore, while dynamic allocation offers greater flexibility, it too is finite.

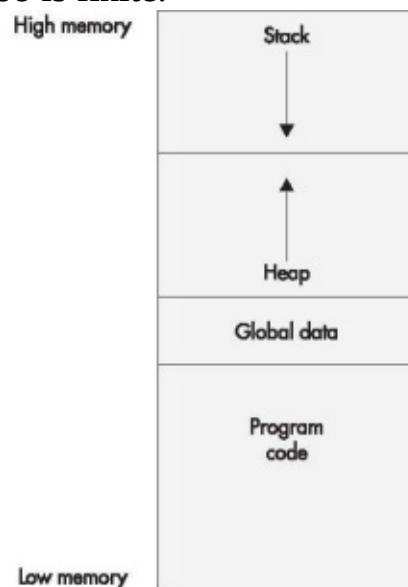


Figure 12-1 A conceptual view of memory usage in a C++ program

C++ provides two dynamic allocation operators: `new` and `delete`. The `new` operator allocates memory and returns a pointer to the start of it. The `delete` operator frees memory previously allocated using `new`. The general forms of `new` and `delete` are shown here:

```
p_var = new type; delete p_var;
```

Here, `p_var` is a pointer variable that receives a pointer to memory that is large enough to hold an item of type `type`.

Since the heap is finite, it can become exhausted. If there is insufficient available memory to fill an allocation request, then `new` will fail and a `bad_alloc` exception will be generated. This exception is

27	

defined in the header `<new>`. Your program should handle this exception and take appropriate action if a failure occurs. If this exception is not handled by your program, then your program will be terminated.

The actions of `new` on failure as just described are specified by Standard C++. The trouble is that some older compilers will implement `new` in a different way. When C++ was first invented, `new` returned a null pointer on failure. Later, this was changed so that `new` throws an exception on failure, as just described. If you are using an older compiler, check your compiler's documentation to see precisely how it implements `new`.

Since Standard C++ specifies that `new` generates an exception on failure, this is the way the code in this book is written. If your compiler handles an allocation failure differently, then you will need to make the appropriate changes.

Here is a program that allocates memory to hold an integer:

```
// Demonstrate new and delete.

#include <iostream>
#include <new>
using namespace std;

int main()
{
    int *p;

    try {
        p = new int; // allocate space for an int
    } catch (bad_alloc xa) {
        cout << "Allocation Failure\n";
        return 1;
    }

    *p = 100;
    cout << "At " << p << " ";
    cout << "is the value " << *p << "\n";

    delete p;

    return 0;
}
```

Allocate an int.

Watch for an allocation failure.

Release the allocated memory.

This program assigns to p an address in the heap that is large enough to hold an integer. It then assigns that memory the value 100 and displays the contents of the memory on the screen. Finally, it frees the dynamically allocated memory.

The delete operator must be used only with a valid pointer previously allocated by using new. Using any other type of pointer with delete is undefined and will almost certainly cause serious problems, such as a system crash.

Initializing Allocated Memory

You can initialize allocated memory to some known value by putting an initializer after the type name in the new statement. Here is the general form of new when an initialization is included:

```
p_var = new var_type (initializer);
```

Of course, the type of the initializer must be compatible with the type of data for which memory is being allocated.

Ask the Expert

Q: I have seen some C++ code that uses the functions `malloc()` and `free()` to handle dynamic allocation. What are these functions?

A: The C language does not support the `new` and `delete` operators. Instead, C uses the functions

`malloc()` and `free()` for dynamic allocation. `malloc()` allocates memory and `free()` releases it. C++ also supports these functions, and you will sometimes see `malloc()` and `free()` used in C++ code. This is especially true if that code has been updated from older C code. However, you should use `new` and `delete` in your code. Not only do `new` and `delete` offer a more convenient method of handling dynamic allocation, but they also prevent several types of errors that are common when working with `malloc()` and `free()`. One other point: Although there is no formal rule that states this, it is best not to mix `new` and `delete` with `malloc()` and `free()` in the same program. There is no guarantee that they are mutually compatible.

This program gives the allocated integer an initial value of 87:

29	

```

// Initialize memory.

#include <iostream>
#include <new>
using namespace std;

int main()
{
    int *p;

    try {
        p = new int (87); // initialize to 87
    } catch (bad_alloc xa) {
        cout << "Allocation Failure\n";
        return 1;
    }

    cout << "At " << p << " ";
    cout << "is the value " << *p << "\n";

    delete p;

    return 0;
}

```

Initialize allocated memory.

Allocating Arrays

You can allocate arrays using new by using this general form:

```
p_var = new array_type [size];
```

Here, size specifies the number of elements in the array. To free an array, use this form of delete:

```
delete [ ] p_var;
```

Here, the [] informs delete that an array is being released. For example, the next program allocates a ten-element integer array:

```
// Allocate an array.
```

```
#include <iostream>
```

```
#include <new>
```

```

using namespace std;

int main()
{
    int *p, i;

    try {
        p = new int [10]; // allocate 10 integer array ←
    } catch (bad_alloc &a) {
        cout << "Allocation Failure\n";
        return 1;
    }

    for(i=0; i<10; i++)
        p[i] = i;

    for(i=0; i<10; i++)
        cout << p[i] << " ";

    delete [] p; // release the array ←
    return 0;
}

```

Allocate an array of int.

Release the array.

Notice the delete statement. As just mentioned, when an array allocated by new is released, delete must be made aware that an array is being freed by using the []. (As you will see in the next section, this is especially important when you are allocating arrays of objects.)

One restriction applies to allocating arrays: They may not be given initial values. That is, you may not specify an initializer when allocating arrays.

Allocating Objects

You can allocate objects dynamically by using new. When you do this, an object is created, and a pointer is returned to it. The dynamically created object acts just like any other object. When it is created, its constructor (if it has one) is called. When the object is freed, its destructor is executed.

Here is a program that creates a class called Rectangle that encapsulates the width and height of a rectangle. Inside main(), an object of type Rectangle is created dynamically. This object is destroyed when the program ends.

```
// Allocate an object.
```

```
#include <iostream>
```



```

#include <new>
using namespace std;

class Rectangle {
    int width;
    int height;
public:
    Rectangle(int w, int h) {
        width = w;
        height = h;
        cout << "Constructing " << width <<
            " by " << height << " rectangle.\n";
    }

    ~Rectangle() {
        cout << "Destructing " << width <<
            " by " << height << " rectangle.\n";
    }

    int area() {
        return width * height;
    }
};

int main()
{
    Rectangle *p;

    try {
        p = new Rectangle(10, 8); ← Allocate a Rectangle object. This
    } catch (bad_alloc xa) {        calls the Rectangle constructor.
        cout << "Allocation Failure\n";
        return 1;
    }

    cout << "Area is " << p->area();

    cout << "\n";

    delete p; ← Release the object. This calls the Rectangle destructor.

    return 0;
}

```

The output is shown here:

Constructing 10 by 8 rectangle.

Area is 80
Destructing 10 by 8 rectangle.

32	


Notice that the arguments to the object's constructor are specified after the type name, just as in other sorts of initializations. Also, because `p` contains a pointer to an object, the arrow operator (rather than the dot operator) is used to call `area()`.

You can allocate arrays of objects, but there is one catch. Since no array allocated by `new` can have an initializer, you must make sure that if the class defines constructors, one will be parameterless. If you don't, the C++ compiler will not find a matching constructor when you attempt to allocate the array and will not compile your program.

In this version of the preceding program, a parameterless constructor is added so that an array of `Rectangle` objects can be allocated. Also added is the function `set()`, which sets the dimensions of each rectangle.

```
// Allocate an array of objects.

#include <iostream>
#include <new>
using namespace std;

class Rectangle {
    int width;
    int height;
public:
    Rectangle() {  Add a parameterless constructor.
        width = height = 0;
        cout << "Constructing " << width <<
            " by " << height << " rectangle.\n";
    }

    Rectangle(int w, int h) {
        width = w;
        height = h;
        cout << "Constructing " << width <<
            " by " << height << " rectangle.\n";
    }

    ~Rectangle() {
        cout << "Destructing " << width <<
            " by " << height << " rectangle.\n";
    }
}
```

33	

```

void set(int w, int h) { ←———— Add the set() function.
    width = w;
    height = h;
}

int area() {
    return width * height;
}
};

int main()
{
    Rectangle *p;

    try {
        p = new Rectangle [3];
    } catch (bad_alloc xa) {
        cout << "Allocation Failure\n";
        return 1;
    }

    cout << "\n";

    p[0].set(3, 4);
    p[1].set(10, 8);
    p[2].set(5, 6);

    for(int i=0; i < 3; i++)
        cout << "Area is " << p[i].area() << endl;

    cout << "\n";

    delete [] p; ←———— This calls the destructor for
                                each object in the array.

    return 0;
}

```

The output from this program is shown here:

```

Constructing 0 by 0 rectangle.
Constructing 0 by 0 rectangle.
Constructing 0 by 0 rectangle.

Area is 12
Area is 80
Area is 30
Destructing 5 by 6 rectangle.
Destructing 10 by 8 rectangle.
Destructing 3 by 4 rectangle.

```


34	

Because the pointer `p` is released using `delete []`, the destructor for each object in the array is executed, as the output shows. Also, notice that because `p` is indexed as an array, the dot operator is used to access members of `Rectangle`.



What operator allocates memory? What operator releases memory?

What happens if an allocation request cannot be fulfilled?

Can memory be initialized when it is allocated?

CRITICAL SKILL 12.5: Namespaces

Namespaces were briefly described in Module 1. Here they are examined in detail. The purpose of a namespace is to localize the names of identifiers to avoid name collisions. In the C++ programming environment, there has been an explosion of variable, function, and class names. Prior to the invention of namespaces, all of these names competed for slots in the global namespace and many conflicts arose. For example, if your program defined a function called `toupper()`, it could (depending upon its parameter list) override the standard library function `toupper()`, because both names would be stored in the global namespace. Name collision problems were compounded when two or more third-party libraries were used by the same program. In this case, it was possible—even likely—that a name defined by one library would conflict with the same name defined by the other library. The situation can be particularly troublesome for class names. For example, if your program defines a class called `Stack` and a library used by your program defines a class by the same name, a conflict will arise.

The creation of the namespace keyword was a response to these problems. Because it localizes the visibility of names declared within it, a namespace allows the same name to be used in different contexts without conflicts arising. Perhaps the most noticeable beneficiary of namespace is the C++ standard library. Prior to namespace, the entire C++ library was defined within the global namespace (which was, of course, the only namespace). Since the addition of namespace, the C++ library is now defined within its own namespace, called `std`, which reduces the chance of name collisions. You can also create your own namespaces within your program to localize the visibility of any names that you think may cause conflicts. This is especially important if you are creating class or function libraries.

Namespace Fundamentals

The namespace keyword allows you to partition the global namespace by creating a declarative region.

In essence, a namespace defines a scope. The general form of namespace is shown here:


```
namespace name { // declarations }
```

35	

Anything defined within a namespace statement is within the scope of that namespace.

Here is an example of a namespace. It localizes the names used to implement a simple countdown counter class. In the namespace are defined the counter class, which implements the counter, and the variables upperbound and lowerbound, which contain the upper and

```
// Demonstrate a namespace.
```

```
namespace CounterNameSpace {  Create a namespace called  
CounterNameSpace.  
    int upperbound;  
    int lowerbound;  
  
    class counter {  
        int count;  
    public:  
        counter(int n) {  
            if(n <= upperbound) count = n;  
            else count = upperbound;  
        }  
  
        void reset(int n) {  
            if(n <= upperbound) count = n;  
        }  
  
        int run() {  
            if(count > lowerbound) return count--;  
            else return lowerbound;  
        }  
    };  
};
```

Here, upperbound, lowerbound, and the class counter are part of the scope defined by the CounterNameSpace namespace.

Inside a namespace, identifiers declared within that namespace can be referred to directly, without any namespace qualification. For example, within CounterNameSpace, the run() function can refer directly to lowerbound in the statement

```
if(count > lowerbound) return count--;
```

However, since namespace defines a scope, you need to use the scope resolution operator to refer to objects declared within a namespace from outside that namespace. For example, to assign the value 10 to upperbound from code outside CounterNameSpace, you must use this statement:

```
CounterNameSpace::upperbound = 10;
```

Or, to declare an object of type counter from outside CounterNameSpace, you will use a statement like this:

```
CounterNameSpace::counter ob;
```

In general, to access a member of a namespace from outside its namespace, precede the member's name with the name of the namespace followed by the scope resolution operator.

Here is a program that demonstrates the use of the CounterNameSpace:

```
// Demonstrate a namespace.
```

```
#include <iostream>
using namespace std;
```

```
namespace CounterNameSpace {
    int upperbound;
    int lowerbound;

    class counter {
        int count;
    public:
        counter(int n) {
            if(n <= upperbound) count = n;
            else count = upperbound;
        }

        void reset(int n) {
            if(n <= upperbound) count = n;
        }

        int run() {
            if(count > lowerbound) return count--;
            else return lowerbound;
        }
    };
}
```

```
int main()
{
    CounterNameSpace::upperbound = 100;
    CounterNameSpace::lowerbound = 0;

    CounterNameSpace::counter ob1(10);

    int i;

    do {
        i = ob1.run();
        cout << i << " ";
    } while(i > CounterNameSpace::lowerbound);
    cout << endl;

    CounterNameSpace::counter ob2(20);
```

Explicitly refer to members of **CounterNameSpace**. Note the use of the scope resolution operator.

37	

```

do {
    i = ob2.run();
    cout << i << " ";
} while(i > CounterNameSpace::lowerbound);
cout << endl;

ob2.reset(100);
CounterNameSpace::lowerbound = 90;
do {
    i = ob2.run();
    cout << i << " ";
} while(i > CounterNameSpace::lowerbound);

return 0;
}

```

Notice that the declaration of a counter object and the references to upperbound and lowerbound are qualified by CounterNameSpace. However, once an object of type counter has been declared, it is not necessary to further qualify it or any of its members. Thus, ob1.run() can be called directly; the namespace has already been resolved.

There can be more than one namespace declaration of the same name. In this case, the namespaces are additive. This allows a namespace to be split over several files or even separated within the same file. For example:


```
namespace NS { int i;  
  
}  
  
// ...  
  
namespace NS { int j;  
  
}
```

Here, NS is split into two pieces, but the contents of each piece are still within the same namespace, that is, NS. One last point: Namespaces can be nested. That is, one namespace can be declared within another.

using

If your program includes frequent references to the members of a namespace, having to specify the namespace and the scope resolution operator each time you need to refer to one quickly becomes

38	

tedious. The using statement was invented to alleviate this problem. The using statement has these two general forms:

```
using namespace name;
```

```
using name::member;
```

In the first form, name specifies the name of the namespace you want to access. All of the members defined within the specified namespace are brought into view (that is, they become part of the current namespace) and may be used without qualification. In the second form, only a specific member of the namespace is made visible. For example, assuming CounterNameSpace as just shown, the following using statements and assignments are valid:

```
using CounterNameSpace::lowerbound; // only lowerbound is visible  
lowerbound = 10; // OK because lowerbound is visible
```

```
using namespace CounterNameSpace; // all members are visible  
upperbound = 100; // OK because all members are now visible
```

The following program illustrates using by reworking the counter example from the

```

// Demonstrate using.

#include <iostream>
using namespace std;

namespace CounterNameSpace {
    int upperbound;
    int lowerbound;

    class counter {
        int count;
    public:
        counter(int n) {
            if(n <= upperbound) count = n;
            else count = upperbound;
        }

        void reset(int n) {
            if(n <= upperbound) count = n;
        }

        int run() {
            if(count > lowerbound) return count--;
            else return lowerbound;
        }
    };
}

```

39	

```

int main()
{
    // use only upperbound from CounterNameSpace
    using CounterNameSpace::upperbound; ← Use a specific member of
                                          CounterNameSpace.

    // now, no qualification needed to set upperbound
    upperbound = 100;
    // qualification still needed for lowerbound, etc.
    CounterNameSpace::lowerbound = 0;
    CounterNameSpace::counter ob1(10);
    int i;

    do {
        i = ob1.run();
        cout << i << " ";
    } while(i > CounterNameSpace::lowerbound);
    cout << endl;

    // Now, use entire CounterNameSpace
    using namespace CounterNameSpace; ← Use the entire
                                          CounterNameSpace.

    counter ob2(20);

    do {
        i = ob2.run();
        cout << i << " ";
    } while(i > lowerbound);
    cout << endl;

    ob2.reset(100);
    lowerbound = 90;
    do {
        i = ob2.run();
        cout << i << " ";
    } while(i > lowerbound);

    return 0;
}

```

The program illustrates one other important point: using one namespace does not override another. When you bring a namespace into view, it simply adds its names to whatever other namespaces are currently in effect. Thus, by the end of the program, both `std` and `CounterNameSpace` have been added to the global namespace.

Unnamed Namespaces

There is a special type of namespace, called an unnamed namespace, that allows you to create identifiers that are unique within a file. It has this general form:

40	
----	--



```
namespace {  
  
// declarations }
```

Unnamed namespaces allow you to establish unique identifiers that are known only within the scope of a single file. That is, within the file that contains the unnamed namespace, the members of that namespace may be used directly, without qualification. But outside the file, the identifiers are unknown. As mentioned earlier in this book, one way to restrict the scope of a global name to the file in which it is declared, is to declare it as static. While the use of static global declarations is still allowed in C++, a better way to accomplish this is to use an unnamed namespace.

The std Namespace

Standard C++ defines its entire library in its own namespace called `std`. This is the reason that most of the programs in this book have included the following statement:

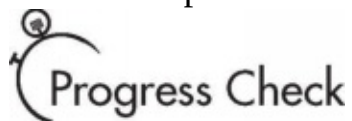
```
using namespace std;
```

This causes the `std` namespace to be brought into the current namespace, which gives you direct access to the names of the functions and classes defined within the library without having to qualify each one with `std::`.

Of course, you can explicitly qualify each name with `std::` if you like. For example, you could explicitly qualify `cout` like this:

```
std::cout << "Explicitly qualify cout with std.";
```

You may not want to bring the standard C++ library into the global namespace if your program will be making only limited use of it, or if doing so will cause name conflicts. However, if your program contains hundreds of references to library names, then including `std` in the current namespace is far easier than qualifying each name individually.



What is a namespace? What keyword creates one?

Are namespaces additive?

What does using do?

41	

CRITICAL SKILL 12.6: static Class Members

You learned about the keyword `static` in Module 7 when it was used to modify local and global variable declarations. In addition to those uses, `static` can be applied to members of a class. Both variables and function members can be declared `static`. Each is described here.

static Member Variables

When you precede a member variable's declaration with `static`, you are telling the compiler that only one copy of that variable will exist and that all objects of the class will share that variable. Unlike regular data members, individual copies of a static member variable are not made for each object. No matter how many objects of a class are created, only one copy of a static data member exists. Thus, all objects of that class use that same variable. All static variables are initialized to zero if no other initialization is specified. When you declare a static data member within a class, you are not defining it. Instead, you must provide a global definition for it elsewhere, outside the class. This is done by redeclaring the static variable using the scope resolution operator to identify which class it belongs to. This causes storage to be allocated for the static variable. Here is an example that uses a static member:

```
// Use a static instance variable.

#include <iostream>
using namespace std;

class ShareVar {
    static int num;
public:
    void setnum(int i) { num = i; };
    void shownum() { cout << num << " "; }
};

int ShareVar::num; // define num

int main()
{
    ShareVar a, b;

    a.shownum(); // prints 0
    b.shownum(); // prints 0

    a.setnum(10); // set static num to 10
}
```

Declare a **static** data member. It will be shared by all instances of **ShareVar**.

Define the **static** data member.

42	

```

a.shownum(); // prints 10
b.shownum(); // also prints 10

return 0;
}

```

The output is shown here:

```
0 0 10 10
```

In the program, notice that the static integer num is both declared inside the ShareVar class and defined as a global variable. As stated earlier, this is necessary because the declaration of num inside ShareVar does not allocate storage for the variable. C++ initializes num to 0 since no other initialization is given. This is why the first calls to shownum() both display 0. Next, object a sets num to 10. Then both a and b use shownum() to display its value. Because there is only one copy of num shared by a and b, both calls to shownum() display 10.

When a static variable is public, it can be referred to directly through its class name, without reference to any specific object. It can also be referred to through an object. For example:

```
// Refer to static variable through its class name.

#include <iostream>
using namespace std;

class Test {
public:
    static int num;
    void shownum() { cout << num << endl; }
};

int Test::num; // define num

int main()
{
    Test a, b;

    // Set num through its class name.
    Test::num = 100; ← Refer to num through
                        its class name Test.


    a.shownum(); // prints 100
    b.shownum(); // prints 100

    // Set num through an object.
```

43	

```
a.num = 200;
a.shownum(); // prints 200
b.shownum(); // prints 200

return 0;
}
```



Refer to num through an object.

Notice how the value of num is set using its class name in this line:

```
Test::num = 100;
```

It is also accessible through an object, as in this line:

```
a.num = 200;
```

Either approach is valid.

static Member Functions

It is also possible for a member function to be declared as static, but this usage is not common. A member function declared as static can access only other static members of its class. (Of course, a static member function may access non-static global data and functions.) A static member function does not have a this pointer. Virtual static member functions are not allowed. Also, it cannot be declared as const or volatile. A static member function can be invoked by an object of its class, or it can be called independent of any object, using the class name and the scope resolution operator. For example, consider this program. It defines a static variable called count that keeps count of the number of objects currently in existence.


```
// Demonstrate a static member function.

#include <iostream>
using namespace std;

class Test {
    static int count;
public:

    Test() {
        count++;
        cout << "Constructing object " <<
            count << endl;
    }
};
```

```

}

~Test() {
    cout << "Destroying object " <<
        count << endl;
    count--;
}

static int numObjects() { return count; }
};

int Test::count;

int main() {
    Test a, b, c;

    cout << "There are now " <<
        Test::numObjects() <<
        " in existence.\n\n";

    Test *p = new Test();

    cout << "After allocating a Test object, " <<
        "there are now " <<
        Test::numObjects() <<
        " in existence.\n\n";

    delete p;
    cout << "After deleting an object, " <<
        "there are now " <<
        a.numObjects() <<
        " in existence.\n\n";

    return 0;
}

```

A static member function

The output from the program is shown here:

```
Constructing object 1
Constructing object 2
Constructing object 3
There are now 3 in existence.

Constructing object 4
After allocating a Test object, there are now 4 in existence.
Destroying object 4
After deleting an object, there are now 3 in existence.

Destroying object 3
Destroying object 2
Destroying object 1
```

45	

In the program, notice how the static function `numObjects()` is called. In the first two calls, it is called through its class name using this syntax:

```
Test::numObjects()
```

In the third call, it is invoked using the normal, dot operator syntax on an object.

CRITICAL SKILL 12.7: Runtime Type Identification (RTTI)

Runtime type information may be new to you because it is not found in non-polymorphic languages, such as C or traditional BASIC. In non-polymorphic languages there is no need for runtime type information, because the type of each object is known at compile time (that is, when the program is written). However, in polymorphic languages such as C++, there can be situations in which the type of an object is unknown at compile time because the precise nature of that object is not determined until the program is executed. As you know, C++ implements polymorphism through the use of class hierarchies, virtual functions, and base class pointers. A base class pointer can be used to point to objects of the base class or to any object derived from that base. Thus, it is not always possible to know in advance what type of object will be pointed to by a base pointer at any given moment. This determination must be made at runtime, using runtime type identification.

To obtain an object's type, use `typeid`. You must include the header `<typeinfo>` in order to use `typeid`. Its most commonly used form is shown here:

```
typeid(object)
```

Here, `object` is the object whose type you will be obtaining. It may be of any type, including the built-in types and class types that you create. `typeid` returns a reference to an object of type `type_info` that describes the type of object.

The `type_info` class defines the following public members:

```
bool operator==(const type_info &ob); bool operator!=(const type_info
```

&ob); bool before(const type_info &ob); const char *name();

The overloaded == and != provide for the comparison of types. The before() function returns true if the invoking object is before the object used as a parameter in collation order. (This function is mostly for internal use only. Its return value has nothing to do with inheritance or class hierarchies.) The name() function returns a pointer to the name of the type.

Here is a simple example that uses typeid:

46	

```

// A simple example that uses typeid.

#include <iostream>
#include <typeinfo>
using namespace std;

class MyClass {
    // ...
};

int main()
{
    int i, j;
    float f;
    MyClass ob;

    cout << "The type of i is: " << typeid(i).name();
    cout << endl;
    cout << "The type of f is: " << typeid(f).name();
    cout << endl;
    cout << "The type of ob is: " << typeid(ob).name();
    cout << "\n\n";

    if(typeid(i) == typeid(j))
        cout << "The types of i and j are the same\n";

    if(typeid(i) != typeid(f))
        cout << "The types of i and f are not the same\n";

    return 0;
}

```

Use `typeid` to obtain the type of an object at runtime.



The output produced by this program is shown here:

```

The type of i is: int
The type of f is: float
The type of ob is: class MyClass
The types of i and j are the same
The types of i and f are not the same

```

Perhaps the most important use of typeid occurs when it is applied through a pointer of a polymorphic base class (that is, a class that includes at least one virtual function). In this case, it will automatically return the type of the actual object being pointed to, which may be a base class object or an object derived from that base. (Remember, a base class pointer can point to objects of the base class or of any class derived from that base.) Thus, using typeid, you can determine at runtime the type of the object that is being pointed to by a base class pointer. The following program demonstrates this

principle:

47	

```

// An example that uses typeid on a polymorphic class hierarchy

#include <iostream>
#include <typeinfo>
using namespace std;

class Base {
    virtual void f() {}; // make Base polymorphic
    // ...
};

class Derived1: public Base {
    // ...
};

class Derived2: public Base {
    // ...
};

int main()
{
    Base *p, baseob;
    Derived1 ob1;
    Derived2 ob2;

    p = &baseob;
    cout << "p is pointing to an object of type ";
    cout << typeid(*p).name() << endl;

    p = &ob1;
    cout << "p is pointing to an object of type ";
    cout << typeid(*p).name() << endl;

    p = &ob2;
    cout << "p is pointing to an object of type ";
    cout << typeid(*p).name() << endl;

    return 0;
}

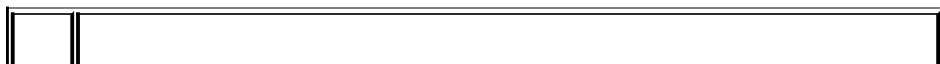
```

The output produced by this program is shown here:

p is pointing to an object of type class Base

p is pointing to an object of type class Derived1 p is pointing to an object of
type class Derived2

When typeid is applied to a base class pointer of a polymorphic type, the
type of object pointed to will be determined at runtime, as the output shows.



In all cases, when typeid is applied to a pointer of a non-polymorphic class hierarchy, then the base type of the pointer is obtained. That is, no determination of what that pointer is actually pointing to is made. As an experiment, comment-out the virtual function f() in Base and observe the results. As you will see, the type of each object will be Base because that is the type of the pointer.

Since typeid is commonly applied to a dereferenced pointer (that is, one to which the * operator has been applied), a special exception has been created to handle the situation in which the pointer being dereferenced is null. In this case, typeid throws a bad_typeid exception.

References to an object of a polymorphic class hierarchy work the same as pointers. When typeid is applied to a reference to an object of a polymorphic class, it will return the type of the object actually being referred to, which may be of a derived type. The circumstance where you will most often make use of this feature is when objects are passed to functions by reference.

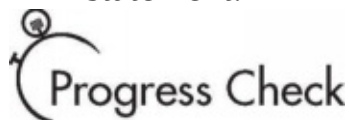
There is a second form of typeid that takes a type name as its argument. This form is shown here:

typeid(type-name)

For example, the following statement is perfectly acceptable:

```
cout << typeid(int).name();
```

The main use of this form of typeid is to obtain a type_info object that describes the specified type so that it can be used in a type comparison statement.



What makes a static member variable unique?

What does typeid do?

What type of object does typeid return?

CRITICAL SKILL 12.8: The Casting Operators

C++ defines five casting operators. The first is the traditional-style cast described earlier in this book. It

has been part of C++ from the start. The remaining four were added a few years ago. They are

`dynamic_cast`, `const_cast`, `reinterpret_cast`, and `static_cast`. These operators give you additional control

over how casting takes place. Each is examined briefly here.

`dynamic_cast`

49	

Perhaps the most important of the additional casting operators is the `dynamic_cast`. The `dynamic_cast` performs a runtime cast that verifies the validity of a cast. If at the time `dynamic_cast` is executed, the cast is invalid, then the cast fails. The general form of `dynamic_cast` is shown here:

`dynamic_cast<target-type> (expr)`

Here, `target-type` specifies the target type of the cast, and `expr` is the expression being cast into the new type. The target type must be a pointer or reference type, and the expression being cast must evaluate to a pointer or reference. Thus, `dynamic_cast` can be used to cast one type of pointer into another or one type of reference into another.

The purpose of `dynamic_cast` is to perform casts on polymorphic types. For example, given two polymorphic classes `B` and `D`, with `D` derived from `B`, a `dynamic_cast` can always cast a `D*` pointer into a `B*` pointer. This is because a base pointer can always point to a derived object. But a `dynamic_cast` can cast a `B*` pointer into a `D*` pointer only if the object being pointed to actually is a `D` object. In general, `dynamic_cast` will succeed if the pointer (or reference) being cast is pointing to (or referring to) either an object of the target type or an object derived from the target type. Otherwise, the cast will fail. If the cast fails, then `dynamic_cast` evaluates to null if the cast involves pointers. If a `dynamic_cast` on reference types fails, a `bad_cast` exception is thrown.

Here is a simple example. Assume that `Base` is a polymorphic class and that `Derived` is derived from `Base`.

```
Base *bp, b_ob;  
Derived *dp, d_ob;  
  
bp = &d_ob; // base pointer points to Derived object  
dp = dynamic_cast<Derived *> (bp); // cast to derived pointer OK  
if(dp) cout << "Cast OK";
```

Here, the cast from the base pointer `bp` to the derived pointer `dp` works

because bp is actually pointing to a Derived object. Thus, this fragment displays Cast OK. But in the next fragment, the cast fails because bp is pointing to a Base object, and it is illegal to cast a base object into a derived object.

```
bp = &b_ob; // base pointer points to Base object
dp = dynamic_cast<Derived *> (bp); // error
if(!dp) cout << "Cast Fails";
```

Because the cast fails, this fragment displays Cast Fails.

const_cast

The const_cast operator is used to explicitly override const and/or volatile in a cast. The target type must be the same as the source type, except for the alteration of its const or volatile attributes. The most common use of const_cast is to remove const-ness. The general form of const_cast is shown here:

const_cast<type> (expr)

50	

Here, type specifies the target type of the cast, and expr is the expression being cast into the new type.

It must be stressed that the use of `const_cast` to cast away const-ness is a potentially dangerous feature.

Use it with care.

One other point: Only `const_cast` can cast away const-ness. That is, `dynamic_cast`, `static_cast`, and `reinterpret_cast` cannot alter the const-ness of an object.

`static_cast`

The `static_cast` operator performs a non-polymorphic cast. It can be used for any standard conversion. No runtime checks are performed. Thus, the `static_cast` operator is essentially a substitute for the original cast operator. Its general form is

```
static_cast<type> (expr)
```

Here, type specifies the target type of the cast, and expr is the expression being cast into the new type.

`reinterpret_cast`

The `reinterpret_cast` operator converts one type into a fundamentally different type. For example, it can change a pointer into an integer and an integer into a pointer. It can also be used for casting inherently incompatible pointer types. Its general form is

```
reinterpret_cast<type> (expr)
```

Here, type specifies the target type of the cast, and expr is the expression being cast into the new type.

What Next?

The purpose of this book is to teach the core elements of the language. These are the features and techniques of C++ that are used in everyday programming. With the knowledge you now have, you can begin writing

real-world, professional-quality programs. However, C++ is a very rich language, and it contains many advanced features that you will still want to master, including:

The Standard Template Library (STL)

Explicit constructors

Conversion functions

const member functions and the mutable keyword

The asm keyword

Overloading the array indexing operator [], the function call operator (), and the dynamic allocation operators, new and delete

Of the preceding, perhaps the most important is the Standard Template Library. It is a library of template classes that provide off-the-shelf solutions to a variety of common data-storage tasks. For

example, the STL defines generic data structures, such as queues, stacks, and lists, which you can use in your programs.

You will also want to study the C++ function library. It contains a wide array of routines that will simplify the creation of your programs.

To continue your study of C++, I suggest reading my book C++: The Complete Reference, published by Osborne/McGraw-Hill, Berkeley, California. It covers all of the preceding, and much, much more. You now have sufficient knowledge to make full use of this in-depth C++ guide.

Module 12 Mastery Check

Explain how try, catch, and throw work together to support exception handling.

How must the catch list be organized when catching exceptions of both base and derived classes?

Show how to specify that a MyExcpt exception can be thrown out of a function called func() that returns void.

Define an exception for the generic Queue class shown in Project 12-1. Have Queue throw this exception when an overflow or underflow occurs. Demonstrate its use.

What is a generic function, and what keyword is used to create one?

Create generic versions of the quicksort() and qs() functions shown in Project 5-1. Demonstrate their use.

Using the Sample class shown here, create a queue of three Sample objects using the generic Queue shown in Project 12-1:

```
class Sample {
    int id;
public:
    Sample() { id = 0; }
    Sample(int x) { id = x; }
    void show() { cout << id << endl; }
};
```

Rework your answer to question 7 so that the Sample objects stored in the queue are dynamically allocated.

Show how to declare a namespace called RobotMotion.

What namespace contains the C++ standard library?

Can a static member function access the non-static data of a class?

What operator obtains the type of an object at runtime?

To determine the validity of a polymorphic cast at runtime, what casting operator do you use?

What does `const_cast` do?

On your own, try putting the Queue class from Project 12-1 in its own namespace called `QueueCode`, and into its own file called `Queue.cpp`. Then rework the `main()` function so that it uses a `using` statement to bring `QueueCode` into view.

Continue to learn about C++. It is the most powerful computer language currently available. Mastering it puts you in an elite league of programmers.

